EARLY MACINTOSH TECHNICAL INFORMATION

INSIDE MACINTOSH
QUICKDRAW: A PROGRAMMER'S GUIDE

COMMENT

VERY EARLY INSIDE MACINTOSH QUICKDRAW CHAPTER SHOWING QUICKDRAW'S EARLY STRUCTURE FROM 1982 AND ALSO EDITING BY CAROLINE ROSE OF CHRIS ESPINOSA'S SOMEWHAT ROUGH DRAFT

AUTHOR

APPLE COMPUTER

DATE

15 FEBRUARY 1982

SOURCE

DAVID T CRAIG • JANUARY 2004

QUIKDRAW.1

Convention needed - whether to capitalize types (e.g. BitMap vs BitMap)

VHSelect
WordPtr

MACINTOSH PUBLICATIONS

QuickDraw: A Programmer's Guide /QUICK.DRAW/QUIKDRAW

See Also:

Modification History:			
	First Draft	C. Espinosa	11/27/81
	Revised and Edited	C. Espinosa	2/15/82

ABSTRACT

This document describes the QuickDraw graphics package, heart of the Macintosh User Interface ToolBox routines. It describes the conceptual and physical data types used by QuickDraw, the procedures and functions available in QuickDraw, and the details of preparing, compiling, and linking a program for use on the Macintosh.

decide on meaning "solid" - filled in with any pattern? yes

See p62 for reference to "MacGraf"

(QD was originally called Lisa Graf) - DTC/24Sep15

FIG 1, 13, 14, 16-19, 113
F 10-12,

All contents are copyright Apple Computer, Inc.

QUIKDRAW

TABLE OF CONTENTS

3 Introduction to QuickDraw
5 About This Manual
6 The Mathematical Foundation of QuickDraw
7 Points
8 Rectangles
1Ø Regions
11 Graphic Objects
12 The Bit Image
13 Patterns; the bitMap
16 Cursors
17 The grafPort
21 Coordinates in GrafPorts
23 Character Fonts
25 General Discussion of Drawing
28 Transfer modes
29 Description of QuickDraw Procedures
29 GrafPort Procedures
33 Cursor-Handling Procedures
34 Pen and Drawing Procedures
36 Character Drawing Procedures
38 Calculations with Rectangles
4Ø Graphic Operations on Rectangles
42 Graphic operations on Ovals
43 Graphic Operations on Rounded-Corner Rectangles
45 Calculations with Regions
49 Graphic Operations on Regions
51 Miscellaneous Utilities
53 Using QuickDraw from Assembly Language
57 Summary of QuickDraw
62 Default Values

QUIKDRAW.2

INTRODUCTION TO QUICKDRAW

QuickDraw is a set of graphics procedures, functions, and data types that allow a Pascal or assembly-language programmer of Macintosh to perform highly complex graphic operations very easily and very quickly.

This document introduces you to QuickDraw and its programmatic interface. It covers the graphic concepts behind QuickDraw, as well as the technical details of the data types, procedures, and functions you will use in your programs. It also describes the fundamentals of preparing, compiling, and linking a program to run on Macintosh.

What QuickDraw Can Do

QuickDraw allows you to divide the Macintosh screen into a number of individual areas, and within each area you can draw many things:-

- Straight lines of any length and width;
- Rectangles, either solid or hollow;
- Circles and ovals, also either solid or hollow;
- Rectangles with rounded corners, solid or hollow;
- Any other arbitrary shape or collection of shapes, again either solid or hollow;
- Images defined dot-by-dot and stored in a file or in the program itself;
- Text characters in a number of proportionally-spaced fonts, with variations that include emboldening, italicising, shadowing, and letterspacing.

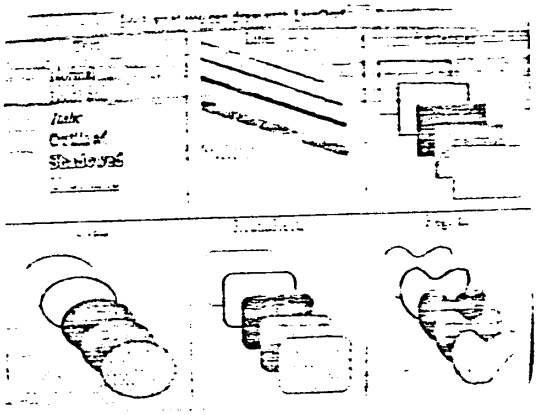


Figure 1. Samples of QuickDraw's Abilities

In addition, QuickDraw has some other abilities that you won't find in many other graphics packages. These abilities take care of most of the "housekeeping" — the trivial but time-consuming overhead that's necessary to keep things ~~straightened up but are really bothersome to do yourself.~~ Among the~~se~~ are:

- The ability to define many distinct "ports" on the screen, each with its own complete drawing environment — its own coordinate system, drawing location, character set, location on the screen, etc. You can switch from one drawing environment to another with just one command.

- Full and complete "clipping" to arbitrary ~~regions~~ areas. It's like a super-duper coloring book that won't let you color outside of the lines. You don't have to worry about accidentally drawing over something else on the screen, or drawing off the screen and trashing memory.

- Off-screen drawing. Anything you can draw on the screen, you can draw into an off-screen buffer, so you can prepare an image for an output device without disturbing the screen, or you can prepare a picture and move it onto the screen very quickly.

And QuickDraw lives up to its name! It is very fast. The speed and responsiveness of the Macintosh user interface is due primarily to the speed of the QuickDraw package. You can do good-quality animation, fast interactive graphics, and complex yet speedy text displays using the full features of QuickDraw.

How To Use QuickDraw

QuickDraw has no user interface of its own; you must write and compile (or assemble) a Pascal (or assembly-language) program that includes the proper QuickDraw calls, link the resulting object code with the QuickDraw code, and execute the linked object file.

Some programming models are supplied at the end of this manual; they show the structure of a properly organized QuickDraw program. What's best for beginners is to obtain a machine-readable version of one of these programs (see the Macintosh software coordinator), read through the ~~code~~ text, and using the superstructure of the program as a "shell", modify it to suit your own purposes. Once you get the hang of writing programs inside the presupplied shell, you can work on changing the shell itself.

In the final Macintoshes, QuickDraw will be stored permanently in the ROM memory (right now, it resides in high RAM). All access is made through an indirection table in low RAM. When you write a program that uses QuickDraw, you link it with this indirection table. Each time you call a QuickDraw procedure or function, or load a predefined constant, the request goes through the table into QuickDraw. You'll never access any QuickDraw address directly, nor will you have to code constant addresses into your program. The linker will make sure all address references get straightened out.

QuickDraw can be used from either Pascal or 68000 machine language. To be clear, concise, and more intuitive, this manual gives all examples in their Pascal form; a section near the end describes the details of the machine language interface to QuickDraw.

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Keyboard input, mouse input, and larger user-interface constructs such as windows and menus are implemented in separate packages that use QuickDraw but are linked in as separate units. You don't need these units in order to use QuickDraw; however, you'll probably want to get the documentation for each of the above and learn how to use them with your Macintosh programs.

ABOUT THIS MANUAL

We assume that you are ~~[illegible]~~ familiar with programming in Pascal and assembly language; this graphics package is for programmers, not end users.

The manual begins by stepping back a little and looking at the mathematical concepts that form the foundation for QuickDraw: coordinate planes, points, and rectangles. The mathematics are then associated with graphic entities that can be seen on the screen. Once you understand these concepts, read on about the graphic entities based

on those concepts: how the mathematical world of planes and rectangles is translated into the physical phenomena of light and shadow.

Then comes some discussion of how to use some of the miscellaneous parts of QuickDraw, such as using multiple graphics ports at once, and manipulating character fonts. Following this discussion is a summary of the basic drawing process. This tells you how to structure your programs so that everything happens in the right order and in the right way. This is a particularly important section: it's got a lot of helpful hints on how to build programs that work right the first time.

Finally, there's the detailed, blow-by-blow description of every QuickDraw procedure and function, their parameters, calling protocol, effects, side effects, etc. all the business that you'll refer to any time you write a program for Macintosh. It's a lengthy section, because there are a lot of QuickDraw calls. But the calls and the parameters are summarized in an appendix for quick reference once you've learned what they do.

THE MATHEMATICAL FOUNDATION OF QUICKDRAW

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region.

The Coordinate Plane

All information about position, location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid:

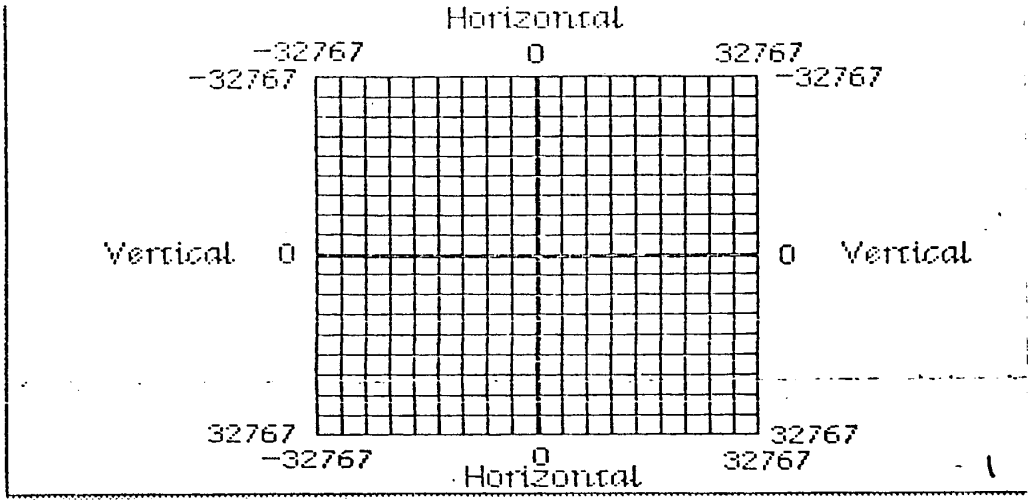


Figure 2. Coordinate Plane

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers; and
- All grid lines are infinitely thin.

These concepts are important! First, they mean that the QuickDraw plane is finite, not infinite (although it is very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates range from -32768 to +32767 also. (An auxiliary package is available that maps real Cartesian space, with X, Y, and Z coordinates, onto QuickDraw's 2D integer coordinate system.)

two-dimensional

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that coordinates are infinitely thin (and a couple more related concepts), you'll never have "endpoint paranoia"--the confusion that results from not knowing whether that last dot is included in the line or not.

Points

On the coordinate plane are 4,294,967,296 unique points; each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of this grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate system.

I don't recall I got the connection

The coordinate origin (Ø,Ø) is in the middle of the grid. Horizontal coordinates increase as you move left to right, and vertical

coordinates increase as you move top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point into a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has this structure:

```
TYPE VHSelect = (v,h);
     Point    = RECORD CASE INTEGER OF

                0:  (v : INTEGER;
                     h : INTEGER);

                1:  (vh: ARRAY[VHSelect] OF INTEGER)

                END;
```

The variant allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable Fermin were declared to be of type Point, the references

Fermin.v
Fermin.vh[v]
Fermin.h
Fermin.vh[h]

would all refer to the coordinate parts of the point.

Rectangles

Any two points can define the top left and bottom right corners of a rectangle. As these points are infinitely small, the borders of the rectangle are infinitely thin.

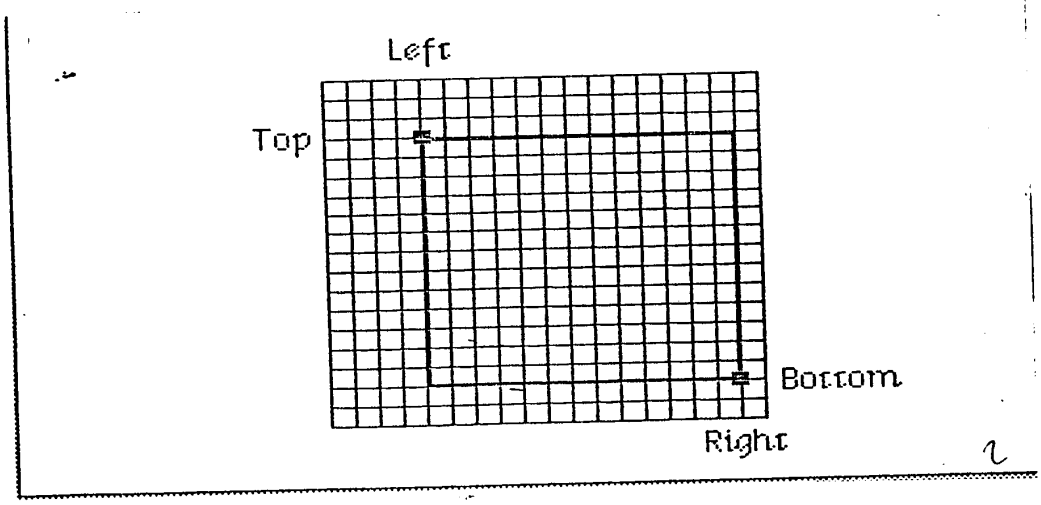


Figure 3. A Rectangle

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphic entities, and to specify the locations and sizes for various drawing commands. All these uses of rectangles are described in the section "Graphical Operations on Rectangles". QuickDraw also allows you to perform many mathematical calculations on rectangles: changing their sizes, shifting them around, etc.

(hand)
Remember that rectangles, like points, are mathematical concepts that have no direct representation on the screen. The association between these conceptual elements and their physical representations is made by a bitMap, described below.

The data type for rectangles is called the Rect, and consists of four integers or two points:

```
TYPE Rect = RECORD CASE INTEGER OF

        0 : (top    : INTEGER;
             left   : INTEGER;
             bottom: INTEGER;
             right  : INTEGER);

        1 : (topLeft : Point;
             botRight: Point)

        END;
```

Once again, the record variant allows you to access a variable of type Rect either as four coordinate boundaries or as two diagonally opposing corner points. Combined with the record variants for points, all of the following references to the Rect named Train are legal:

```
Train.....{Rect type}

Train.topLeft.....Train.botRight.....{Point type}

Train.top.....Train.left.....{INTEGER type}
Train.topLeft.v.....Train.topLeft.h.....{INTEGER "}
Train.topLeft.vh[v].....Train.topLeft.vh[h].....INTEGER "

Train.bottom.....Train.right.....INTEGER
Train.botRight.v.....Train.botRight.h.....INTEGER
Train.botRight.vh[v].....Train.botRight.vh[h].....INTEGER
```

(eye)
If the bottom coordinate of a rectangle is not greater than the top, or the right coordinate is not greater than the left, then the rectangle is not a rectangle. All operations that use that rectangle will have no effect whatsoever.

Regions

Unlike most graphics packages that can manipulate only simple geometric structures (usually rectilinear, at that), QuickDraw has the unique and amazing ability to gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. This remarkable feature will not only make your standard programs simpler and faster, but will let you perform operations that would otherwise be nearly impossible; it is fundamental to the Macintosh user interface.

You define a region by drawing lines, ~~or framing~~ rectangles, ovals, and rounded-corner rectangles, or other regions. The outline of a region ~~is mathematically defined~~ as one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle.

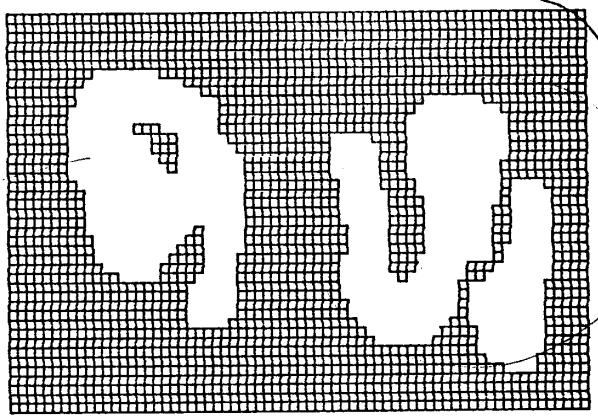


Figure 4. Regions

Because a region can be any arbitrary area on the coordinate plane, it takes a variable amount of information to store the outline of a region. The data structure for a region, therefore, is a variable-length entity with two fixed fields at the beginning, followed by a variable-length data field:

```
TYPE Region = RECORD
               rgnSize : INTEGER;
               rgnBBox : Rect;
               {optional region definition data}
               END;
```

The rgnSize field contains the size, in bytes, of the region variable. The rgnBBox field is a rectangle which completely encloses the region. For example, if the variable Lumbar is a region, then the field Lumbar.RgnBBox.top contains the minimum vertical coordinate of all points in the region.

The simplest region is a rectangle; in this case, the rgnBBox field serves to define the entire region, and there is no optional region data. For rectangular regions (or empty regions), the rgnSize field contains 10.

As regions are of variable size, they are stored dynamically on the heap, and the Memory Manager moves them around as their sizes change. Being dynamic, a region can be accessed only through a pointer; but when a region is moved, all pointers referring to it must be updated. For this reason, all regions are accessed through handles, which point to one master pointer which in turn points to the region.

```
TYPE RgnPtr    = ^Region;
     RgnHandle = ^RgnPtr;
```

When the Memory Manager relocates a region's data in memory, it updates only the RgnPtr master pointer to that region. The references through the master pointer can find the region's new home, but those references that pointed directly to the region's previous position in memory now point at dead bits. To access individual fields of a region, use the region handle and double indirection:

Steve^.rgnSize	Size of region whose handle is Steve
Steve^.rgnBBox	Boundary box of same region
Steve^.rgnBBox	Syntactically incorrect; will not compile if Steve is a rgnHandle

Regions are created by a QuickDraw function which allocates space for the region, creates a master pointer, and returns a RgnHandle. When you're done with a region you can dispose of it with another QuickDraw call, which frees up the space used by the region. Only these calls allocate or deallocate regions; DO NOT use the Pascal procedure NEW to create a new region!

Many calculations can be performed on regions. A region can be "grown" or "shrunk", and given any two regions, QuickDraw can find their union, intersection, difference, and exclusive-OR; it can also determine whether a given point or rectangle intersects a given region, etc. There is of course a set of graphical operations on regions to draw or outline them on the screen, as

The outline of a region is specified originally by graphical operations, which are described in the section "Pen and Drawing Procedures".

GRAPHIC OBJECTS

Coordinate planes, points, rectangles, and regions are all good mathematical models, but they aren't really graphic elements--they don't have a direct physical appearance. Some graphic entities that do

have a direct graphic interpretation are the bit image, pattern, cursor, and bitMap. This section describes the data structure of these graphic entities and how they relate to the mathematical constructs described above. The next section begins to describe the graphic operations that you can perform with QuickDraw to draw pictures.

The Bit Image

A bit image is a collection of bits in memory which have a rectilinear representation. Take a collection of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit Ø of the highest-numbered word is on the far right. Then take this array of bits and divide it, on byte boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on top and the last row is on bottom.

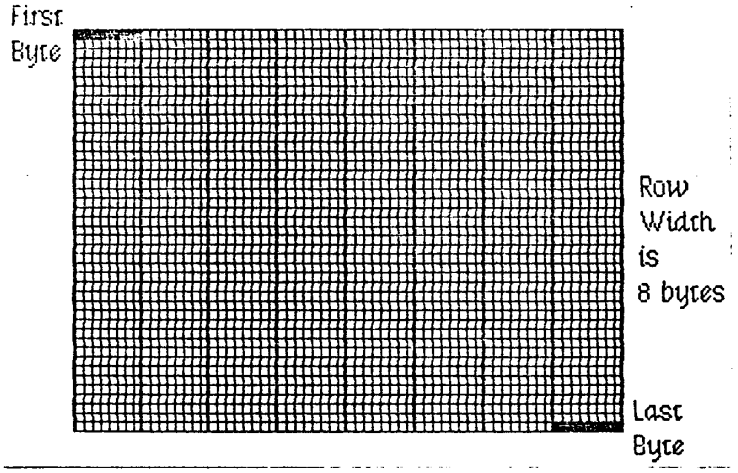


Figure 5. A Bit Image

The result is a matrix like the above--rows and columns of bits, with each row containing the same number of bytes. The number of bytes in each row of the bit image is called the row width of that image.

A bit image can be stored in any static or dynamic variable, and can be of any length that is a multiple of the row width.

The Macintosh screen itself is one large visible bit image. The upper 12K bytes of memory is displayed as a matrix of 122,880 pixels" on the screeen, each bit corresponding to one pixel. If a bit's value is Ø, its pixel is white; if the bit's value is 1, the pixel is black.

The screen is 256 pixels tall and 384 pixels wide, and the row width of its bit image is 48 bytes. Each pixel on the screen is square; the aspect ratio of the display is 3:2 (h:v).

(hand)
As each pixel on the screen represents one bit in a bit image, whenever this document says "bit", you can substitute "pixel".

Patterns

A pattern is a 64-bit image, organized as an 8-by-8-bit square, which is used to define a repeating image or tone. Patterns can be used to draw with, to fill an area, or function as the source for a bitwise operation.

Patterns are drawn aligned to the current local coordinate system, so that drawing adjacent or overlapping areas with the same pattern produces a continuous, coordinated pattern. QuickDraw provides predefined patterns for White, Black, Gray, LtGray, and DkGray. Any other 64-bit variable or constant can be used as a pattern, too. The data type definition for a pattern is as follows:

```
TYPE Pattern = PACKED ARRAY[0..7] OF 0..255
```

The row width of a pattern is 1 byte.

The bitMap

When you combine the physical entity of a bit image with the conceptual entities of the coordinate grid and rectangle, you get a bitMap. A bitMap is a descriptor that has three parts: a pointer to a bit image, the row width (in bytes) of that image, and a bounding rectangle which gives the bitMap both its dimensions and a coordinate system. Notice that a "bitMap", as we will use the term, does not actually include the bits themselves: a bitMap is a descriptor containing a pointer to a bit image.

Because a bitMap points to a bit image, there can be several bitMaps extant simultaneously that all impose different coordinate systems on the same bit image. This important feature is explained more fully in "Coordinates in grafPorts", below.

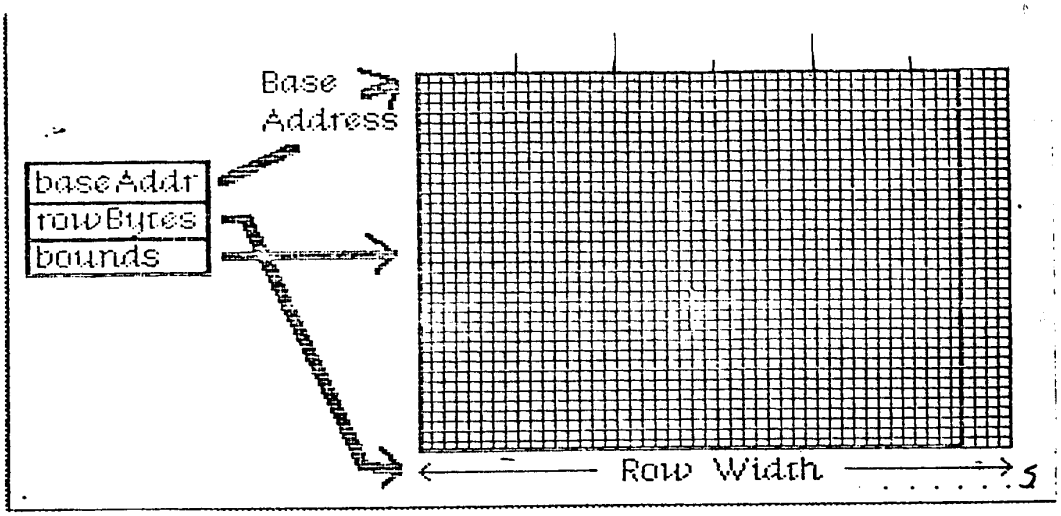


Figure 6. A bitMap

The data structure of a bitMap follows:

```
bitMap = RECORD
             baseAddr : WordPtr;
             rowBytes : INTEGER;
             bounds   : Rect
         END;
```

The baseAddr field is a pointer to the beginning of the bit image in memory, and the rowBytes field is the number of bytes in each row of the image. Both of these should always be even: a bitMap should always begin on a word boundary, and contain an integral number of words in each row.

The bounds field is a bounding rectangle that both encloses the active area of the bit image and imposes a coordinate system upon it. The relationship of the bounding rectangle and bit image in a bitMap is simple yet very important. First, a few general rules:

- Bits in a bitMap fall between points on the coordinate grid.
- A rectangle divides a bit image into two sets of bits: those bits inside the rectangle and those outside the rectangle.
- A rectangle that is H points wide and V points tall encloses exactly (H*V) bits.

The top left corner of the bounding rectangle is aligned around the first bit in the bitMap. The width of the rectangle determines how many bits of one row are logically owned by the bitMap; the relationship

Map.rowBytes >= (Map.bounds.right-Map.bounds.left)

must always be true. The height of the rectangle determines how many rows of the image are logically owned by the bitMap; the relationship

```
SIZEOF(Map.baseAddr^) >= (Map.bounds.bottom-Map.bounds.top)
                         * Map.rowBytes
```

must always be true to ensure that the number of bits in the logical bitMap area is not larger than the number of bits in the bit image.

Normally, the bounding rectangle completely encloses the bit image: the width of the bounding rectangle is equal to the number of bits in one row of the image, and the height of the rectangle is equal to the number of rows in the image. If the rectangle is smaller than the dimensions of the image, the least significant bits in each row, as well as the last rows in the image, are not affected by any operations on the bitMap.

The bitMap also imposes a coordinate system on the image. Because bits fall between coordinate points, the coordinate system assigns integer values to the lines that border and separate bits, not to the bit positions themselves. If a bitMap is assigned the bounding rectangle (10,-5) (35,-5) (10,8) (35,8), the bottom right bit in the image will be between horizontal coordinates 34 and 35, and between vertical coordinates 7 and 8.

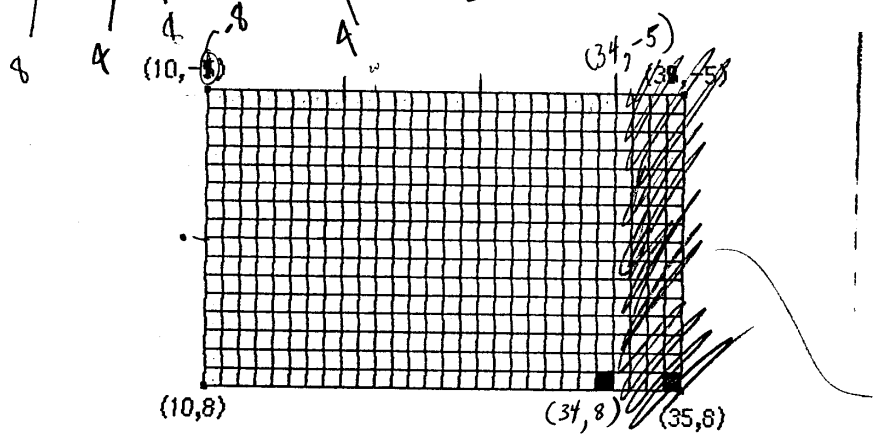


Figure 7. Coordinates and bitMaps

Cursors

A cursor is a small image that appears on the screen and is controlled by the mouse. Cursors are defined as a 256-bit image, a 16-by-16-bit square. The row width of a cursor is 2 bytes.

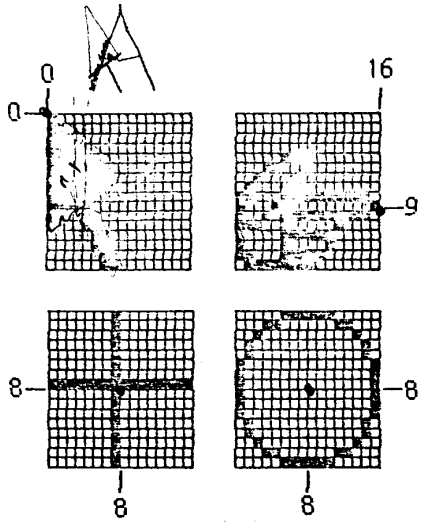


Figure 8. Cursors

A cursor has three fields: a 16-word data field that contains the image itself, a 16-word mask field that contains information about the screen appearance of each bit of the cursor, and a "hotSpot" point that aligns the cursor with the position of the mouse.

```
Cursor = RECORD
             data = ARRAY[Ø..15] OF INTEGER;
             mask = ARRAY[Ø..15] OF INTEGER;
             hotSpot = Point
         END;
```

The data for the cursor must begin on a word boundary.

The cursor appears on the screen as a 16-by-16-bit square. Each bit of the square is ~~controlled~~ by two corresponding bits in the data and mask, and, if the mask bit is Ø, by the pixel "under" the cursor (the one already on the screen in the same position as this bit of the cursor):

Data	Mask	~~Result~~ Pixel on screen
Ø	1	~~Pixel in cursor appears~~ White
1	1	~~Pixel in cursor appears~~ Black
Ø	Ø	Same as Pixel under cursor ~~shows through~~
1	Ø	Inverse of Pixel under cursor ~~is inverted~~

The hot spot aligns a point in the image (not a bit, a point!) with the mouse position. Imagine the rectangle with corners (Ø,Ø) and (~~15,15~~) framing the image; the hotSpot is defined in this coordinate system. A hotSpot of (Ø,Ø) is at the top left of the image; a hotSpot of (~~7,7~~) is in the exact center of the image. Whenever you move the

mouse, the low-level interrupt-driven mouse routines move the cursor's hot spot to be aligned with the new mouse position.

(hand)
The mouse position is always, unalterably linked to the cursor position. You can't reposition the cursor through software: the only control you have is whether it is visible or not, and what shape it will assume. Think of it as being hard-wired: if the cursor is visible, it always follows the mouse over the full size of the screen.

QuickDraw supplies one predefined cursor, a pointing arrow, named Arrow.

The grafPort

A grafPort is a complete drawing environment that defines how and where graphic operations will have their effect. A grafPort is like a logical output device: it contains all the information about one instance of graphic output that is kept separate from all other instances. You can have many grafPorts open at once, and each one will have its own coordinate system, drawing pattern, background pattern, pen size and location, character font, size, and style, and bitMap into which drawing takes place; one statement will instantly switch you from one port to another. GrafPorts are the structures on which we build windows, which are fundamental to the Macintosh "overlapping folders" user interface.

A grafPort is a dynamic data structure with ten fields:

```
TYPE grafPtr  = ^grafPort;
     grafPort = RECORD
                    portBits : BitMap;
                    portRect : Rect;
                    visRgn   : RgnHandle;
                    clipRgn  : RgnHandle;
                    bkPat    : Pattern;
                    chStyle  : CharStyle;
                    pnLoc    : Point;
                    pnSize   : Point;
                    pnMode   : INTEGER;
                    pnPat    : Pattern
                END;
```

All QuickDraw operations refer to grafPorts via grafPtrs. You create a grafPort with the Pascal procedure NEW and use the resultant pointer in calls to QuickDraw. You could, of course, declare a static VAR of type grafPort, and obtain a pointer to that static structure (with the @ operator), but as most grafPorts will be used dynamically, their data structures should be dynamic also.

2/17/82 Espinosa /QUICK.DRAW/QUIKDRAW.3

(hand)

You can access all fields and sub-fields of a grafPort normally, but you should not store new values directly into them. QuickDraw has procedures for altering all fields of a grafPort, and using these procedures ensures that changing a grafPort produces no unusual side effects.

The portBits field of a grafPort is the bitMap that points to the bit image in memory to be used by the grafPort. All drawing that is done in this grafPort will take place in this bit image. The default bitMap uses the Macintosh video screen as its bit image, with a rowBytes of 48 and has a bounding rectangle of (0,0) (384,256). The bitMap can be changed to indicate a different structure in memory: all graphics procedures work in exactly the same fashion regardless of whether their effects are visible on the screen. A program can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen; or a picture can be developed in an off-screen bitmap before being transferred whole to the screen. The coordinate system of the grafPort can be changed by altering the coordinates of the portBits.bounds rectangle; with a QuickDraw procedure call, you can set arbitrary coordinate systems for each grafPort, even if they all use the same bit image (i.e. the screen).

The portRect field is a rectangle that defines a subset of the bitMap for use by the grafPort. Its coordinates are in the system defined by the portBit bitMap's bounding rectangle. All drawing done by the user or application program is clipped to the inside of this rectangle; the portRect usually defines the "writable" interior area of a folder, document, or other object on the screen.

The visRgn field is manipulated by the Window Manager; users and programmers will normally never change a grafPort's visRgn. It indicates that region (remember, an arbitrary area or set of areas) that is actually visible on the screen. For example, if you move one folder on top of another, the Window Manager logicaly removes the area of overlap from the visRgn of the folder on bottom. When you draw into the bottom folder, whatever's being drawn is clipped to the visRgn so that it doesn't run over onto the top folder. The default visRgn is set to the portRect.

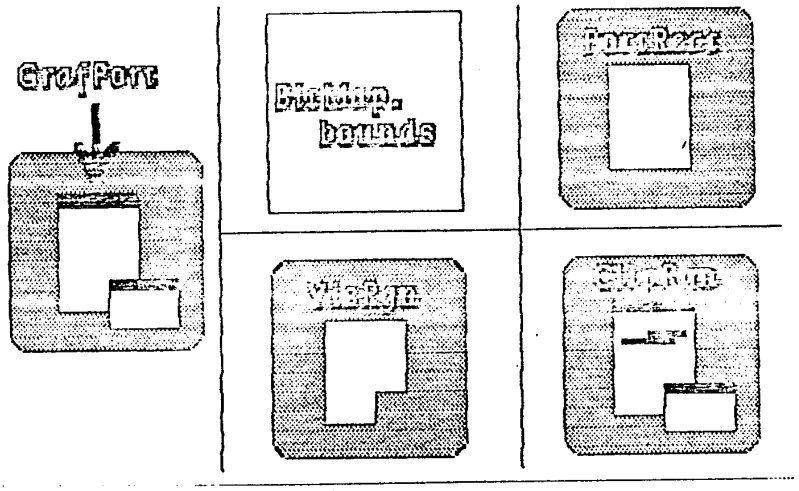


Figure 9. grafPort Regions

The clipRgn is an arbitrary region that you can use to limit drawing to any region within the portRect. If, for example, you want to draw a half circle on the screen, you can set the clipping region to half the square that would enclose the whole circle, and go ahead and draw the whole circle. Only the half within the clipRgn will actually be drawn in the grafPort. The default clipping region is set arbitrarily large, and you have full control over its setting.

The bkPat is a pattern that is used for filling in the background of a picture during erasing or scrolling. This pattern, like all other patterns, is always drawn aligned with the local coordinates of the grafPort.

The chStyle is a record (see below) that determines the font and style of characters drawn in the grafPort.

The last four fields deal with the graphics pen. Each grafPort has one and only one graphics pen, which is used for drawing lines and printing text. The pen has four characteristics: a location, a size, a drawing mode, and a drawing pattern.

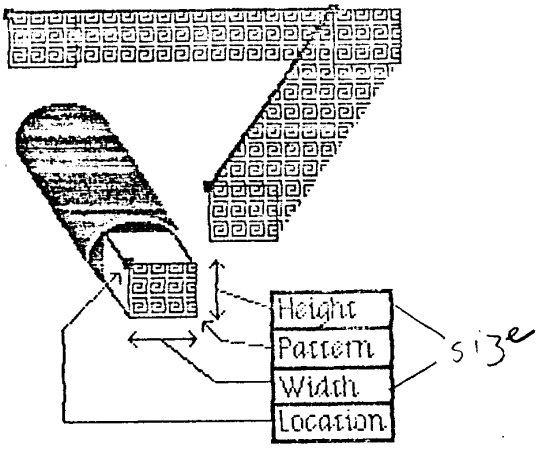


Figure 1Ø. A Graphics Pen

The pen location is a point in the local coordinate system of the grafPort (see below). The pen location can be anywhere on the coordinate plane: there are no restrictions on the movement or placement of the pen. Remember that the pen location is a point on the coordinate plane, not a pixel in a bitMap!

The pen is rectangular in shape, and has a user-definable width and height. The default size is a 1-by-1 square; the width and height can range from (Ø,Ø) to (32767,32767). If either the pen width or the pen height are less than 1, however, the pen will not draw on the screen.

- The pen appears as a rectangle with its upper-left corner at the pen location; it hangs below and to the right of the pen location.

The pnMode and pnPat fields of a grafPort determine how those bits under the pen are affected. The pnPat is a pattern that is used like the "ink" in the pen. The pattern is drawn on the bitMap aligned with the local coordinate system of the grafPort, so that drawing adjacent or overlapping areas in the same pattern produces a continuous, coordinated pattern. Five patterns are predefined (white, black, and three shades of gray); you can create your own tonal or repeating pattern and use it as the pnPat. (A utility procedure, called StuffHex, allows you to fill patterns easily.)

The pnMode field determines how the pen pattern is to affect what's already on the bitMap. When the pen draws, QuickDraw first determines what bits of the bitMap will be affected, and finds their corresponding bits in the pattern. It then does a bit-by-bit evaluation based on the pen mode, which specifies one of eight boolean operations to perform. The resulting bit is placed into its proper place in the bitMap. The pen modes are discussed in "Transfer Modes", below.

The following graphics procedures use the pnSize when drawing:

LineTo	Line
FrameRect	FrameOval
FrameRoundRect	FrameRgn

These procedures, as well as the ones above, also use the pnMode and pnPat when drawing:

PaintRect	PaintOval
PaintRoundRect	PaintRgn

COORDINATES IN GRAFPORTS

Each grafPort has its own local coordinate system. All fields in the grafPort are expressed in these coordinates, and all calculations and actions performed in QuickDraw use the local coordinate system of the currently selected port.

Two things are important to remember:

- Each grafPort maps a portion of a coordinate grid into a similarly-sized portion of a bit image.
- The portBits.bounds field defines the local coordinates for a grafPort.

The interrelationship between the portBits.bounds and portRect rectangles is very important. First, the top left corner of portBits.bounds is always aligned around the first bit in the bit image; the coordinates of that corner "anchor" a point on the grid to that pixel in the bit image. This forms a common reference point for multiple grafPorts using the same bit image (such as the screen); given a portBits.bounds rectangle for each port, you know that their top left corners coincide.

As the portBits.bounds rectangle establishes a coordinate system for the port, the portRect rectangle indicates the section of the coordinate grid (and thus the bit image) will be used for drawing. The portRect usually falls inside the portBits.bounds rectangle, but it is not required to do so.

When a new grafPort is created and initialized, its bitMap is set to point to the Macintosh screen, and both the portBits.bounds and the portRect rectangles are set to 384-by-256 rectangles, with the origin point (0,0) at the top left corner of the screen.

Your local origin point of the grafPort is always the top left corner of the portRect. You can change the local point with the SetOrigin procedure; this recalculates all coordinate systems in the grafPort to

be relative to the new origin point. For example, calling the procedure

```
SetPort(Oakland);
SetOrigin(20,50)
```

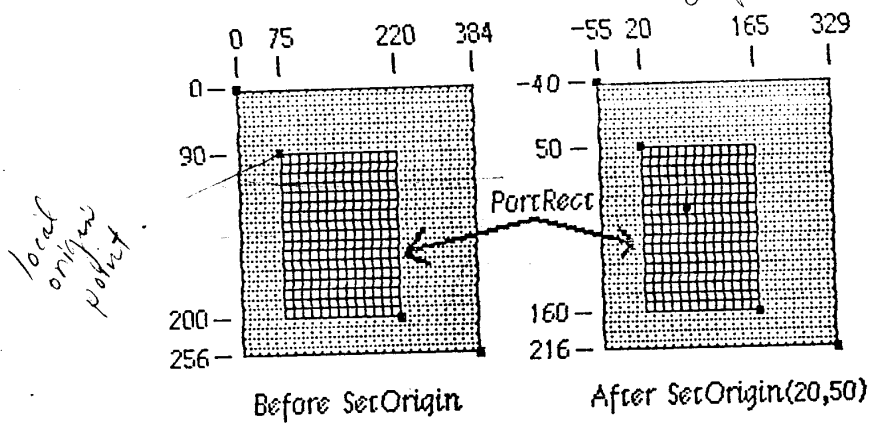


Figure 11. Setting Local Coordinates

will recalculate the coordinate components of these elements:

Oakland^.portBits.bounds Oakland^.portRect

Oakland^.visRgn

All of the above items are always "in sync", so to speak: all calculations, comparisons, or operations that seem right, work right.

Notice that when the local coordinates of a grafPort are offset, the visRgn of that port is offset also, but the clipRgn is not. A good way to think of it is that if a document is being shown inside a grafPort, the document "sticks" to the coordinate system and the port's structure "sticks" to the screen. Scrolling a document upwards inside its port involves keeping all the document's stuff (like the clipRgn) at the same position, while all the port's stuff (like its portBits.bounds, portRect, and visRgn) is moved downwards. Since the top left corner of portBits.bounds is fixed, moving its coordinates down looks like you're moving the document up.

(hand)

All drawing takes place within a grafPort and uses the local coordinates of that grafPort. If you are moving, comparing, or otherwise dealing with mathematical items in different grafPorts (for example, finding the intersection of two regions in two different grafPorts), you must adjust to a common coordinate system before you

MISSING PAGE

QuickDraw first determines what bits of the bitMap will be affected, and does a bit-by-bit comparison based on the mode (described in "Transfer Modes", below). The resultant bits are stored into the bitMap.

(hand)
The character drawing mode should be restricted to either srcOr, srcXor, or srcBic in order to make kerned characters not obliterate each other.

The next four fields control the appearance of the font: emboldening, italicizing, letterspacing, and shadowing. You can apply these either alone or in combination; however, combinations of modes usually look good only for large fonts. All parameters should be positive; never set any of these parameters less than zero!

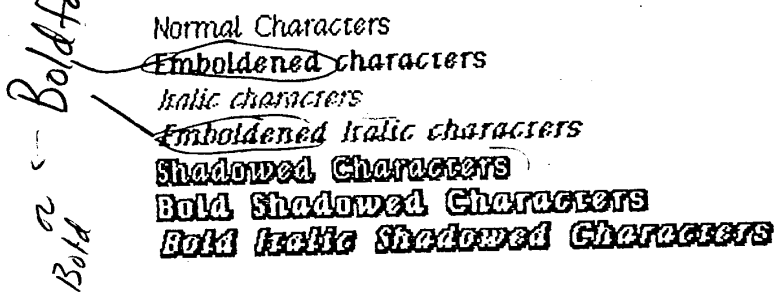


Figure 12. Character Styles

The bold parameter controls linear overstriking of characters: the character is repeatedly "printed" one bit to the right, the number of times specified by the bold field.

The slant parameter adds an italic slant to the characters. A slant of Ø produces characters whose verticals are perpendicular to the baseline; a slant of 8 produces a nice amount of italicising for a 1Ø-point font. The number of dots of horizontal skew per vertical line is determined by this formula:

$$\text{skewdots} := \frac{\text{slant}}{16}$$

Character bits above the baseline are skewed right; bits below the baseline are skewed left.

The extra parameter adds space after the character; the ~~normal~~ extra space is Ø. You can add extra space in order to ~~letterspace characters~~; it's also helpful to add extra space ~~when you embolden~~ or shadow characters. ~~As~~ slanted characters are kerned, it's not ~~usually~~ necessary to give them extra space.

Character shadowing, controlled by the shadow parameter, is like ~~emboldening~~ but makes a hollow, outlined character rather than a solid one. Like ~~emboldening~~, the shadow parameter controls the number of times the character is "overstruck"; reasonable values are Ø, 1, 2, and 3.

GENERAL DISCUSSION OF DRAWING

Drawing occurs:

- Always inside of a grafPort, with the port's coordinate system;
- Usually with the grafPort's pen (that is, pen location, size, and pattern);
- On the grafPort's bitMap; and
- Always clipped to the intersection of the portRect, bitMap bounds, visible region, and clipping region.

Three things can be drawn: lines, shapes, and characters. Shapes include rectangles, ovals, rounded-corner rectangles, and regions, all either solid or framed (hollow).

Lines are defined by two points: the current pen location and a destination location. When drawing a line, QuickDraw draws the top left corner of the pen along the mathematical trajectory from the current location to the destination. The pen hangs below and to the right of the trajectory.

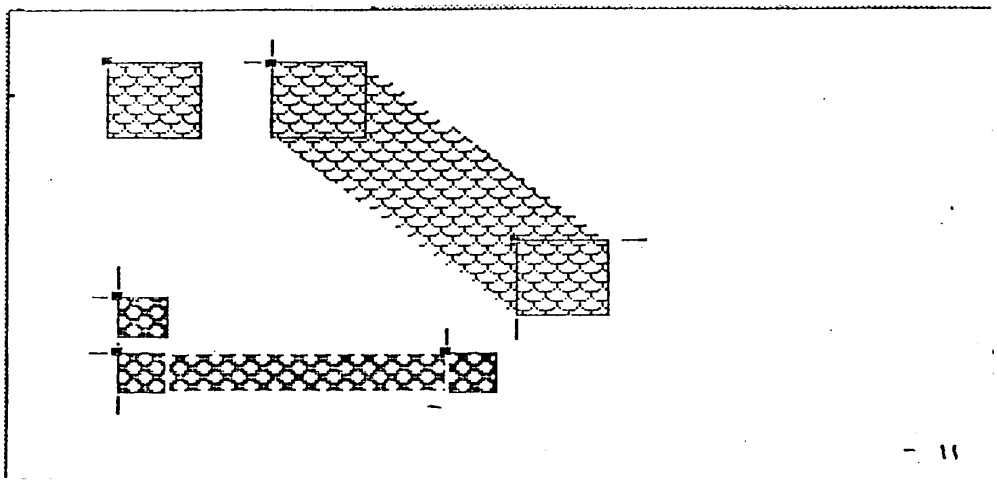


Figure 13. Drawing Lines

(hand)

No mathematical element (such as the pen location) is ever affected by clipping; clipping only determines what appears where on the screen. If you draw a line to a location outside of your grafPort, the pen location will remain there; but only the portion of the line that is inside the port will actually be drawn. This is true for all drawing procedures.

Simple shapes (rectangles, ovals, and rounded-corner rectangles) are defined by two corner points. The shapes always appear inside the mathematical rectangle defined by the two points. A region is defined in a more complex manner, but it too appears only within the rectangle ~~of its bounding box~~.

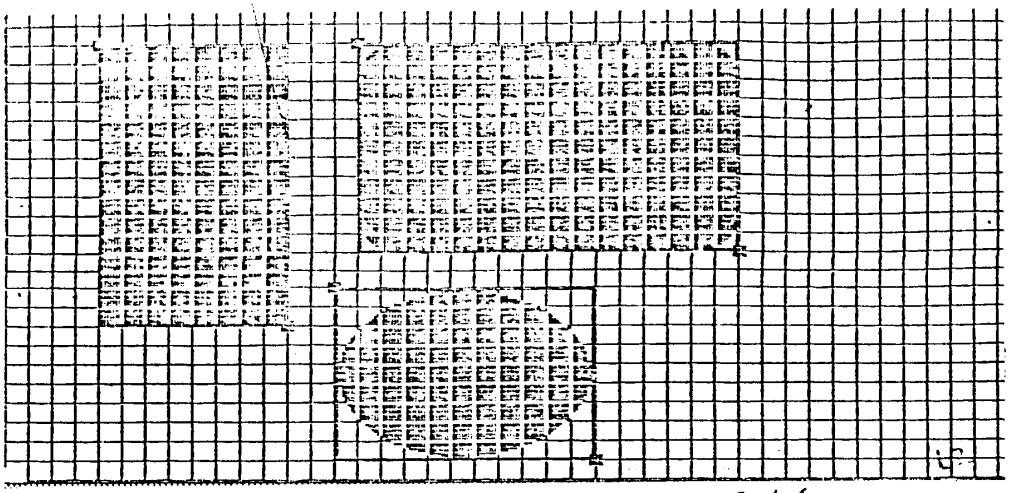


Figure 14. Drawing Shapes

In the case of framed shapes, the outline still appears within the defining rectangle, and the vertical and horizontal thickness of the outline is determined by the pen size:

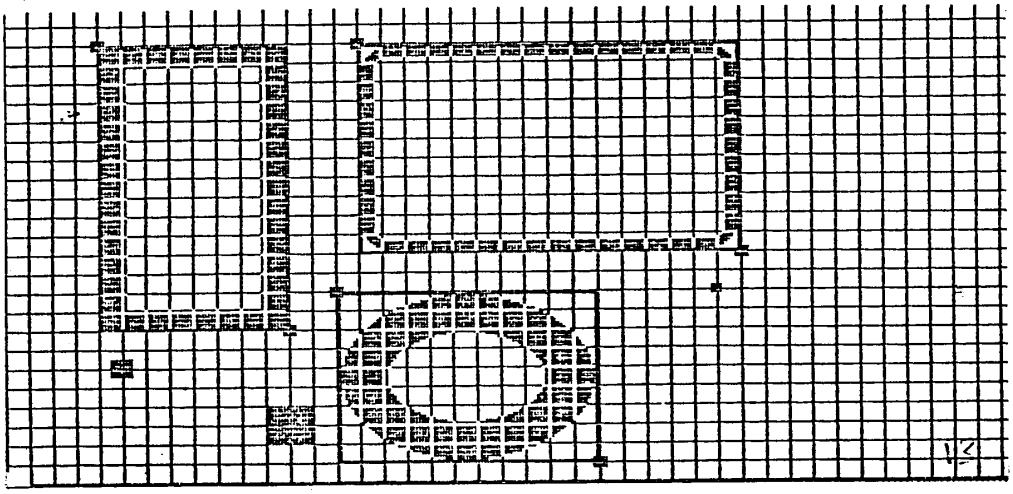


Figure 15. Drawing Framed Shapes

The pen pattern is used to fill in the bits that are affected by the drawing operation. The pen mode defines how those bits are to be affected by directing QuickDraw to apply one of eight boolean operations to the bits in the shape and the corresponding bits on the screen.

Character drawing does not use the pnMode or pnPat, but it does use the ~~pnBits~~. Characters are placed to the right of the current pen location, with their baselines at the pen vertical location, and the pen is moved

to the right a distance equal to the width of the character. No wrap or carriage return is performed automatically.

The method QuickDraw uses in placing characters is controlled by a mode parameter similar to the pen mode. This parameter is explained in "Transfer Modes", below. Clipping of characters is performed in exactly the same manner as all other clipping in QuickDraw.

Transfer Modes

When drawing lines, shapes, or characters, the pnMode (or chStyle.mode) field determines how the drawing is to appear on the screen. For each bit in the item to be drawn, QuickDraw finds the corresponding bit in the bit image, and performs one of sixteen boolean transformations on the pair of bits, and stores the resulting bit into the bit image.

There are four basic operations: Copy, OR, XOR, and BIC. Each of these has a variant in which the drawing bit is inverted before the operation is performed, giving eight operations. These eight are used for drawing with a pattern; there are another eight for transferring one bit image onto another, giving sixteen operations in all. Each mode is defined by name as a constant in QuickDraw.

Transferring a Source Bit Image (XferRect, XferRgn)

Source Transfer Mode		Action for each bit in source: If Black:	If White:
srcCopy	"Paint"	Force Black	Force White
srcOr	"Overlay"	Force Black	Leave Alone
srcXor	"Invert"	Invert	Leave Alone
srcBic	"Erase"	Force White	Leave Alone
notSrcCopy	"Paint"	Force Black	Force White
notSrcOr	"Overlay"	Leave Alone	Force White
notSrcXor	"Invert"	Leave Alone	Invert
notSrcBic	"Erase"	Leave Alone	Force White

Transferring a Pattern (Line, LineTo, Paint, Frame)

Pattern Transfer Mode		Action for each bit in pattern: Black:	White:
patCopy	"Paint"	Force Black	Force White
patOr	"Overlay"	Force Black	Leave Alone
patXor	"Invert"	Invert	Leave Alone
patBic	"Erase"	Force White	Leave Alone
notPatCopy	"Paint"	Force Black	Force White
notPatOr	"Overlay"	Leave Alone	Force White
notPatXor	"Invert"	Leave Alone	Invert
notPatBic	"Erase"	Leave Alone	Force White

The pnMode parameter must be one of the pattern modes; the chStyle.mode parameter must be one of the source modes.

2/17/82 Espinosa /QUICK.DRAW/QUIKDRAW.4

DESCRIPTION OF QUICKDRAW PROCEDURES

The following pages describe ~~each~~ all the functions and procedures in QuickDraw, their parameters, and their operation. They are given in their Pascal form; instructions on how to make procedure and function calls from assembly language appears after the calls.

grafPort Procedures

PROCEDURE InitGraf;

Call once and only once at the beginning of your program to initialize QuickDraw. It initializes the QuickDraw global variables:

Variable	Type	~~Default~~ Setting
white	Pattern	All-white pattern
black	Pattern	All-black pattern
gray	Pattern	50% grey pattern
ltGray	Pattern	33% grey pattern
dkGray	Pattern	66% grey pattern
arrow	Cursor	Pointing arrow cursor
charNormal	charStyle	NO FONT, SrcOr mode, no extras
screenBits	BitMap	Macintosh screen, (0,0,384,256)
thePort	GrafPtr	NIL
~~rgnSave~~	BOOLEAN	False

InitGraf also places the cursor on the screen.

PROCEDURE OpenPort (gp: GrafPtr);

Initializes the selected grafPort and makes it the current port (see SetPort). You must perform an OpenPort before using any grafPort. To use OpenPort, perform a NEW to create a GrafPtr and use this GrafPtr in the OpenPort call.

These default values are set by OpenPort:

Variable	Type	Setting
portBits	BitMap	ScreenBits (see InitGraf)
portRect	Rect	ScreenBits.bounds (0,0,384,256)
visRgn^^	Region	Rectangular, (0,0,384,256)
clipRgn^^	Region	Rectangular, (-30000,-30000,30000,30000)
bkPat	Pattern	White
chStyle	CharStyle	CharNormal
pnLoc	Point	(0,0)
pnSize	Point	(1,1)
pnMode	INTEGER	patCopy
pnPat	Pattern	~~Black~~

MISSING PAGE

The top left corner of the portRect remains at its same location; the width of the portRect is set to width, and the height of the portRect is set to height. In other words, PortSize moves the bottom right corner of the portRect to a position relative to the top left corner.

PortSize does not update the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort: it only changes its width and height. Remember that all drawing occurs only in the intersection of the portRect, the portBits.bounds, the visRgn, and the clipRgn.

PROCEDURE MovePortTo (leftGlobal,topGlobal: INTEGER);

Changes the position of the portRect in the bitMap (read "on the screen") without altering the local coordinate system of the port. THIS DOES NOT AFFECT THE SCREEN; it merely changes the location on the screen at which drawing inside the port will appear.

(hand)
This procedure is normally called only by the Window Manager.

leftGlobal and topGlobal set the distance between the corners of portBits.bounds and the new portRect. For example,

```
MovePortTo(192,128);
```

will move the top left corner of the portRect to the center of the screen (if portBits is the Macintosh screen) regardless of the local coordinate system.

MovePortTo recalculates the coordinates of ~~portBits.bounds~~ but leaves all other coordinates unchanged.

PROCEDURE SetOrigin (h,v: INTEGER);

This changes the local coordinate system of the current grafPort. THIS DOES NOT AFFECT THE SCREEN; it does, however, affect where subsequent drawing and calculation will appear in the port on the screen. It updates the coordinates of the portRect, the portBits.bounds, and the visRgn. All subsequent drawing and calculation commands will use the new coordinate system.

The h and v parameters set the coordinates of the top left corner of the portRect. All other coordinates are calculated from this point. All relative distances among any elements in the port will remain the same; only their absolute local coordinates will change.

(hand)
SetOrigin does not update the coordinates of the clipRgn or the pen; these items stick to the document, inside the port, not the port on the screen.

PROCEDURE SetVisRgn (rgn: RgnHandle);

Declares the region whose handle is supplied in rgn to be the visible region of the current port.

This procedure is normally called only by the Window Manager.

(hand)
Note that SetVisRgn does not make a copy of the object region; it merely saves the RgnHandle in grafPort.visRgn. Thus, any subsequent changes you make to rgn will also affect the visible region of the port.

Be careful never to dispose a region that has been set as the visible region of a grafPort, unless you are also disposing the grafPort itself.

PROCEDURE GetVisRgn (VAR rgn: RgnHandle);

Returns a pointer to the ~~current~~ visible region. It is defined in the coordinate system of the grafPort; if you wish to use this region in another port, you must offset it (see "Coordinate Systems").

PROCEDURE SetClipRgn (rgn: RgnHandle);

Declares the region whose handle is supplied in rgn to be the clipping region of the current port. You can set the clipping region to any arbitrary region, to aid you in drawing inside the grafPort. The default clipRgn is an arbitrarily large rectangle.

(hand)
Note that SetClipRgn does not make a copy of the object region; it merely saves the RgnHandle into grafPort.clipRgn. Thus, any subsequent changes you make to rgn will also affect the clipping region of the port. Be careful never to dispose a region that has been set as the clipping region (unless you are disposing of the grafPort itself).

PROCEDURE GetClipRgn (VAR rgn: RgnHandle);

Returns a handle to the clipping region of the current grafPort. You can use this, for example, to get a handle to use in a region operation such as XorRgn, UnionRgn, OffsetRgn, ~~etc.~~ and

It is defined in the coordinate system of the grafPort; if you wish to use this region in another port, you must offset it (see the LocalToGlobal and GlobalToLocal calls).

↓ pg 52 Miscellaneous Utilities

PROCEDURE ClipRect (r: Rect);

Sets the clipping region of the current grafPort to a rectangle equivalent to the rectangle r. NOTE that this does not change the region handle, but affects the region itself; in combination with SetClipRgn, this may produce unexpected side effects. For example,

```
SetClipRgn(myRegion);
{.....}
ClipRect(anyRect)
```

will not only set the clipping region to anyRect, but will also change myRegion to anyRect.

PROCEDURE BackPat (pat: Pattern);

Sets the background pattern of the current grafPort to pat. The background pattern is used in EraseRect, EraseOval, EraseRoundRect, EraseRgn, and ScrollRect.

Cursor-Handling Procedures

PROCEDURE SetCursor (crsr: Cursor);

Sets the current cursor to the 16-by-16 image in crsr. If the cursor is hidden, it remains hidden and will attain the new appearance when it is uncovered; if the cursor is already visible, it attains the new appearance immediately.

The initial cursor image, when a port is created, is visible on the screen as a north-northwest arrow. There is no way to retrieve the current cursor image.

PROCEDURE HideCursor;

Removes the cursor from the screen, restoring the bits under it. HideCursor also decrements a "cursor-level" variable, which keeps track of the number of times the cursor has been hidden to compensate for nested calls to HideCursor and ShowCursor. Every call to HideCursor should be balanced by a subsequent call to ShowCursor. At initialization, the cursor is not hidden.

PROCEDURE ShowCursor;

Increments the "cursor-level" variable and, if the level becomes positive, displays the cursor on the screen. The level pins at zero, so extra calls to ShowCursor don't hurt. Each call to ShowCursor should balance a previous call to HideCursor.

QuickDraw low-level interrupt-driven routines link the cursor with the mouse position, so that if the cursor level is Ø (visible), the cursor automatically follows the mouse. You don't need to do anything but a ShowCursor to have a cursor track the mouse. There is no way to "disconnect" the cursor from the mouse; you can't force the cursor to a certain position, nor can you easily prevent the cursor from entering a certain area of the screen.

If the cursor has been changed (with SetCursor) while hidden, ShowCursor presents the new cursor.

The initial cursor state is a north-northwest arrow, not hidden.

PROCEDURE CursorVis (visible: BOOLEAN);

A general-purpose cursor showing and hiding procedure: CursorVis(TRUE) simply calls ShowCursor, and CursorVis(FALSE) calls HideCursor.

Pen and Drawing Procedures

The pen and drawing procedures ~~are all dependent upon~~ the coordinate system of the current grafPort. Remember that each grafPort has its own pen; if you draw in one grafPort, change to another, and return to the first, the pen will have remained in the same location.

PROCEDURE PenSize (width,height: INTEGER);

Sets the dimensions of the graphics pen in the current grafPort. All subsequent Line, LineTo, FrameRect, FrameOval, FrameRoundRect, and FrameRgn procedure calls in the same grafPort will use the new pen dimensions.

If either of the pen dimensions ~~are~~ set to nonpositive values, the pen assumes the dimensions (Ø,Ø) and no drawing is performed. The current pen dimensions can be accessed in the variable thePort^.pnSize, which is of type Point (it has components .h and .v). For a discussion of how the pen draws, see the "General Discussion of Drawing" earlier in this document.

PROCEDURE GetPen (VAR pt: Point);

Returns the current pen location, in the local coordinates of the current grafPort.

(eye)

If the coordinate system has been changed with SetOrigin since the last pen motion, the pen will not be in the same location on the screen. The pen "sticks" to the document: moving the document ~~within the screen~~ with SetOrigin carries the pen with it into a new location in the grafPort. It will, however, be in the same location

relative to the coordinates of other items.

PROCEDURE GetPenState (VAR pnState: PenState);

Saves the pen location, size, and pattern into a storage variable, to be restored later with SetPenState (below). This is useful when calling short subroutines that operate in the current port but must change the graphics pen: each such procedure can save the pen's state when it's called, do whatever it needs to do, and restore the previous pen state immediately before returning.

The PenState data type is not really useful for anything except saving the pen's state.

PROCEDURE SetPenState (pnState: PenState);

Sets the pen's location, size, and pattern in the current grafPort to the values stored in pnState. This is usually called at the end of a procedure that has altered the pen parameters and wants to restore them to their state at the beginning of the procedure (see GetPenState, above).

PROCEDURE PenMode (mode: INTEGER);

Sets the transfer mode through which the PenPat is transferred onto the BitMap. The mode may be any one of the pattern modes:

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

If the mode is one of the modes (or non-positive), no drawing is performed. The current pen mode can be obtained as an integer in the variable thePort^.pnMode. The initial pen mode is patCopy, in which the pen pattern is copied directly to the bitMap.

PROCEDURE PenPat (pat: Pattern);

Sets the pattern that is used by the pen in this grafPort. Standard patterns are defined for White, Black, Gray, LtGray, and DkGray; the initial pnPat is Black. The current pen pattern can be obtained in the variable thePort^.pnPat, and this value can be assigned (but not compared!) to any other variable of type Pattern.

PROCEDURE PenNormal;

Resets the initial state of the pen in grafPort:

PnSize	(1,1)
PnMode	patCopy

PnPat Black

The pen location is not changed.

PROCEDURE MoveTo (h,v: INTEGER);

Moves the pen to location (h,v) in the local coordinates of the current grafPort. No drawing is performed.

PROCEDURE Move (dh,dv: INTEGER);

Moves the pen a distance dh horizontally and a distance dv vertically from its previous location. The positive directions are down and right. No drawing is performed.

PROCEDURE LineTo (h,v: INTEGER);

Draws a line from the current pen location to the location specified (in local coordinates) by h and v. The new pen location becomes (h,v) after the line is drawn. See the general discussion on drawing.

If a region is open and being formed, the trajectory of the new line is mathematically added to the region's boundary.

PROCEDURE Line (dh,dv: INTEGER);

Draws a line a distance dh horizontally and a distance dv vertically from its previous location. The positive directions are down and right. The pen location assumes the coordinates of the end of the line. See the general discussion on drawing.

If a region is open and being formed, the trajectory of the new line is mathematically added to the region's boundary.

Character-Drawing Procedures

Each grafPort has its own character style, and all these procedures deal with the character style of the current port.

PROCEDURE DrawChar (ch: CHAR);

Places the selected character to the right of the pen location, and advances the pen the width of the character. If the character is not in the font, the font's "missing symbol" will be drawn; if the font's "missing symbol" is itself missing, expect a crash (in this incarnation of QuickDraw).

PROCEDURE DrawString (s: Str255);

Performs consecutive calls to DrawChar for each character in the supplied string; the string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

(eye)
If the string of characters extends beyond the clipping boundaries of the current grafPort, clipping will be performed as normal, but the pen location will remain to the right of the last character completely or partially drawn, not the last character in the string. This is the only exception to the rule that clipping never affects the pen location.

PROCEDURE DrawText (textBuf: WordPtr; firstByte,byteCount: INTEGER);

Draws text from an arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes, and draws them on the screen in a string. The string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc) is performed by QuickDraw. The pen location ends up to the right of the last character in the string (but see the warning for DrawString, above).

The textBuf parameter, although of type WordPtr, is really a byte pointer; it may be an odd value to point at either byte in a word.

FUNCTION CharWidth (ch: CHAR): INTEGER;

Returns the width of the given character; this value is added to the pen horizontal coordinate whenever the specified character is drawn. The CharWidth includes the effects of the extra space parameter.

FUNCTION StringWidth (s: Str255): INTEGER;

Returns the width of the given text string; this same value is added to the pen horizontal coordinate whenever the specified string is drawn. The StringWidth includes the effects of the extra space parameter; if you change this parameter before actually drawing the string, your width will not be accurate.

(eye)
StringWidth and TextWidth always return the true length of the string as it would be drawn, regardless of clipping. But the actual difference between the beginning and ending pen locations may be affected by clipping (see DrawString and DrawText, above): in such

cases, the distance the pen moves while drawing the string is not the same as the width as given by StringWidth or TextWidth.

```
FUNCTION TextWidth (textBuf: WordPtr; firstByte,byteCount: INTEGER):
            INTEGER;
```

Returns the width of the ~~given~~ text ~~from an~~ arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. The TextWidth includes the effects of the extra space parameter; if you change this parameter before actually drawing the text, your width will not be accurate. (See also the warning for StringWidth, above).

The textBuf parameter, although of type WordPtr, is really a byte pointer; it may be an odd value to point at either byte in a word.

```
PROCEDURE SetChar (chS: CharStyle);
```

Sets the prepared chS to be the character style of the current grafPort. You must initialize all fields of a CharStyle before installing it with SetChar. Calling SetChar(charNormal) will restore the default character font.

After calling SetChar, all previously calcuated character, string, and text widths may become invalid due to a new font or extra space parameter.

The current character style may be retrieved from the variable thePort^.chStyle.

Calculations with Rectangles

(hand)

Calculation procedures are independent of the current coordinate systems; a calculation will operate the same regardless of which grafPort is active.

```
PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom: INTEGER);
```

Utility to assign four boundary coordinates to a rectangle. The result is a rectangle with coordinates (left,top,right,bottom).

This procedure is supplied to help you shorten your ~~source code~~; if you want more readable ~~code~~ at the expense of length, you can assign integers (or points) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method; one's just easier to write, and the other's easier to read.

```
PROCEDURE OffsetRect (VAR r: Rect; dh,dv: INTEGER);
```

"Moves" the coordinates of a rectangle a distance dh horizontally and a distance dv vertically; positive directions are right and down. The rectangle retains its shape and size; only its coordinate locations ~~is~~ are altered.

```
PROCEDURE InsetRect (VAR r: Rect; dh,dv: INTEGER);
```

Shrinks or ~~grows~~ expands a rectangle. The top and bottom are moved towards the center by the amount specified in dv; the left and right sides are moved in by the amount in dh. The effect is to alter the size by 2*dh horizontally and 2*dv vertically, with the rectangle remaining centered in the same place on the coordinate plane. If dh or dv is negative, the appropriate pair of sides is moved outwards instead of inwards.

If either the resultant width or height becomes less than 1, the rectangle is set to the null rectangle (Ø,Ø,Ø,Ø).

```
FUNCTION  SectRect (srcRectA,srcRectB: Rect; VAR dstRect: Rect):
            BOOLEAN;
```

Calculates the rectangle that is the intersection of the two input rectangles, and returns TRUE if they indeed intersect and FALSE if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the ~~B~~itMap.

If the rectangles do not intersect, the destination rectangle is set to (Ø,Ø,Ø,Ø). SectRect works correctly even if one of the source rectangles is ~~used as~~ also the destination.

(hand)

If the rectangles ~~are appropriate to~~ ~~[redacted]~~ different grafPorts, their mathematical intersection (~~according to~~ SectRect) may be different ~~than~~ from their physical intersection on the screen. To adjust the points of a rectangle into another port's coordinate system, see the LocalToGlobal and GlobalToLocal calls.

```
FUNCTION  PtInRect (pt: Point; r: Rect): BOOLEAN;
```

Determines if the pixel below and to the right of the given point is enclosed in the given rectangle, and returns TRUE if so ~~and~~ FALSE if not. (The chosen pixel is the one that would be altered by a Line(Ø,Ø) at the current pen location with a pen size of (1,1).) Both the point and the rectangle must be in the same coordinate system.

Graphical Operations on Rectangles

PROCEDURE PaintRect (r: Rect);

Paints the specified rectangle with the pen pattern and mode. The rectangle on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location.

PROCEDURE FrameRect (r: Rect);

Draws a hollow outline with the pen pattern, mode, and size just inside the specified rectangle. The outline is as wide as the pen width and as tall as the pen height. The outline on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

PROCEDURE EraseRect (r: Rect);

Paints the specified rectangle with the background pattern bkPat. The pnPat and pnMode are ignored; the pen remains at its original location.

PROCEDURE InvertRect (r: Rect);

The bits enclosed by the specified rectangle are inverted: every white bit becomes black and every black bit becomes white. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location.

PROCEDURE FillRect (r: Rect; pat: Pattern);

Paints the specified rectangle with the given pattern. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location.

PROCEDURE ScrollRect (dstRect: Rect; dh,dv: INTEGER; updateRgn: rgnHandle);

ScrollRect alters only those bits inside the dstRect, the visRgn, the clipRgn, the portRect, and the portBits.bounds. The bits in the intersection of all those get shifted a distance of dh horizontally and a distance of dv vertically; positive directions are right and down. Bits that are shifted out of the scroll area are lost; they are neither placed outside the area nor saved off. The background pattern bkPat is

scrolled into the space created by the scroll. In addition, updateRgn is changed to contain all bits that were scrolled in.

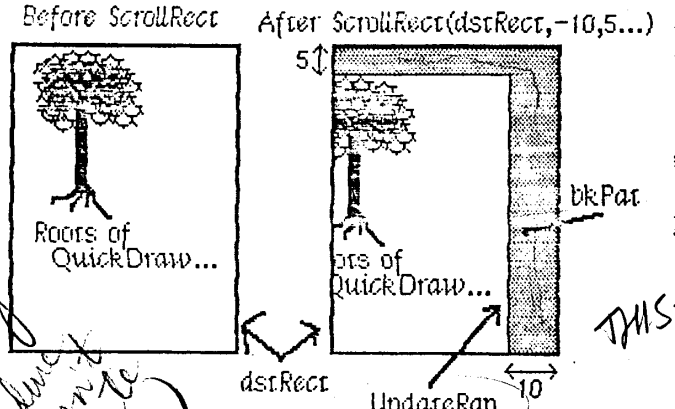


Figure 16. Scrolling

To speed updating after scrolling, you can temporarily set the clipping region to the update region, redraw only the area inside clipRgn^.rgnBBox, then reset the clipping region. This will make QuickDraw refresh only the area of the port that was actually emptied by the scroll. A utility that makes this procedure easier is included in the Window Manager.

```
PROCEDURE XferRect (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
          mode:INTEGER; pat:Pattern);
```

Transfers a bit image between any two bitMaps, or from a pattern into any rectangular area within a bitMap. The transfer may be performed in any of the 16 transfer modes. The transfer is always clipped to the boundary of the destination bitMap, and, if the destination bitMap is the current grafPort's portBits, the transfer is clipped inside the clipRgn, visRgn, and portRect, too.

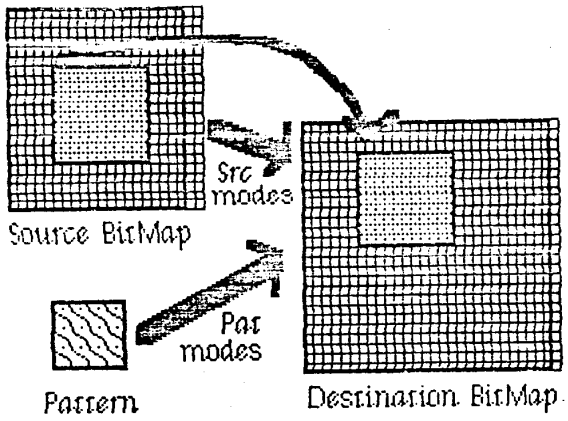


Figure 17. Operation of XferRect

The DstRect coordinates are in terms of the dstBits.bounds coordinate system; the SrcRect coordinates are in terms of the srcBits.bounds coordinates. When copying from the source to the destination, the top left corner of the source rectangle is aligned with the top left corner of the destination rectangle; the size of the source rectangle is ignored.

In the source-using transfer modes, each bit of the source is transferred to the destination according to the rules of that mode, and the pattern is ignored (we suggest you set it to white). The eight source-using modes are as follows:

srcCopy	srcXor	notSrcCopy	notSrcXor
srcOr	srcBic	notSrcOr	notSrcBic

In the pattern-using transfer modes, the specified pattern is transfered into the destination rectangle according to the chosen mode. The source bitMap and source rectangle are not used; we suggest you set these to the destination bitMap and destination rectangle. The eight pattern transfer modes are as follows:

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

These transfer modes are useful for filling a rectangle that is outside of the current grafPort. For filling a rectangle inside the current grafPort, however, the FillRect procedure is much easier to use.

Graphical Operations on Ovals

```
PROCEDURE PaintOval (r: Rect);
```

Paints an oval just inside the specified rectangle with the pen pattern and mode. The oval on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location.

```
PROCEDURE FrameOval (r: Rect);
```

Draws a hollow outline with the pen pattern, mode, and size just inside the oval that fits inside the specified rectangle. The outline is as wide as the pen width and as tall as the pen height. The outline on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

```
PROCEDURE EraseOval (r: Rect);
```

Paints an oval just inside the specified rectangle with the background pattern bkPat. The pnPat and pnMode are ignored; the pen remains at its original location.

```
PROCEDURE InvertOval (r: Rect);
```

The bits enclosed by an oval just inside the specified rectangle are inverted: every white bit becomes black and every black bit becomes white. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location.

```
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

Paints an oval just inside the specified rectangle with the given pattern. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location.

Graphical Operations on Rounded-Corner Rectangles

```
PROCEDURE PaintRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
```

Paints the specified rounded-corner rectangle with the pen pattern and mode. The rounded-corner rectangle on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

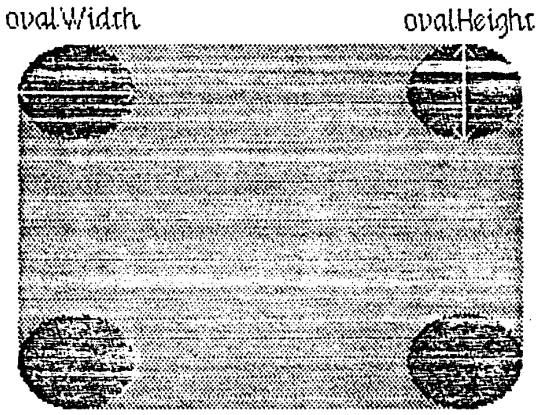


Figure 18. Rounded-Corner Rectangles

PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

Draws a hollow outline with the pen pattern, mode, and size just inside the specified rounded-corner rectangle. The outline is as wide as the pen width and as tall as the pen height. The outline on the bitMap is filled with the pnPat, according to the transfer mode specified by pnMode. The pen remains at its original location. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

If a region is open and being formed, the outside outline of the new rounded-corner rectangle is mathematically added to the region's boundary.

PROCEDURE EraseRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

Paints the specified rounded-corner rectangle with the background pattern bkPat. The pnPat and pnMode are ignored; the pen remains at its original location. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);

The bits enclosed by the specified rounded-corner rectangle are inverted: every white bit becomes black and every black bit becomes white. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

```
PROCEDURE FillRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER; pat:
          Pattern);
```

Paints the specified rounded-corner rectangle with the given pattern. The pnPat, pnMode, and bkPat are all ignored; the pen remains at its original location. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

Calculations with Regions

```
FUNCTION NewRgn: RgnHandle;
```

Creates space for a new, dynamic, variable-size region, initializes it to the empty region, and returns a handle to the new region. Only this procedure creates new regions; all other procedures just alter the size and shape of regions you create.

Never refer to a region without using its handle; regions are too big and awkward to be carried around any other way.

(eye)
You must call NewRgn to initialize a region's handle before that handle can be used in any region drawing or calculation procedure.

```
PROCEDURE DisposeRgn (rgn: RgnHandle);
```

Deallocates space for the region whose handle is supplied, and returns the memory used by the region to the free memory pool. Use this only after you are completely through with a temporary region.

(eye)
NEVER use a region once you have deallocated it, or you will risk being hung by dangling pointers! Beware of SetClipRgn and SetVisRgn; they don't duplicate the region, they only make a copy of the handle. If you deallocate a region that you previously set to be your clipRgn, your clipping goes away and very strange things will occur.

```
PROCEDURE CopyRgn (srcRgn,dstRgn: RgnHandle);
```

Copies the mathematical structure of srcRgn into dstRgn; i.e. it makes a duplicate copy of srcRgn. Once this is done, srcRgn may be altered (even disposed of) without affecting dstRgn.

PROCEDURE SetRectRgn (rgn: RgnHandle; left,top,right,bottom: INTEGER);

Destroys the previous data structure of rgn, then sets the new structure to the rectangle specified by left, top, right, and bottom. This is a good way to obtain a rectangular region, and also to set a region to the null region (Ø,Ø,Ø,Ø).

If the rectangle specified is not rectangular (i.e. left>=right or top>=bottom), the region is set to the null region.

PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);

Destroys the previous data structure of rgn, then sets the new structure to the rectangle specified by r. This is operationally synonymous with SetRectRgn, except the input rectangle is defined by a rectangle, not four boundary coordinates.

PROCEDURE OpenRgn;

OpenRgn tells QuickDraw to allocate temporary space and start saving line segments and framed shapes ~~to~~ later ~~be~~ processed as a region definition. The global BOOLEAN variable RgnSave indicates whether a region is open; flipping RgnSave FALSE disables the collection of lines and frames, and restoring RgnSave TRUE resumes the region definition.

All Line, LineTo, FrameRect, FrameOval, FrameRoundRect, and FrameRgn procedure calls while a region is open affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition, not the pen mode, pattern, size, or location. In fact, it's best to set the penSize to (Ø,Ø) before drawing a region: ~~as~~ the pen hangs below and to the right on a line, drawing lines with even the smallest pen will change bits that do not mathematically belong to the region.

The outline of a region is mathematically defined and infinitely thin, and separates the bitMap into two groups of bits: those within the region and those outside it. A region should consist of one or more closed loops. Each frame itself constitutes a loop; any lines drawn with Line or LineTo should connect with each other or with a frame. Even though the on-screen presentation of a region is clipped to the portRect, etc., the creation of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

(eye)

Do not perform an OpenRgn while another region is already open. All open regions but the most recent will behave strangely.

```
PROCEDURE CloseRgn (dstRgn: RgnHandle);
```

Stops the collection of endpoints and outlines, organizes them into a region definition, and saves the resulting region into the region indicated by dstRgn. You should perform one and only one CloseRgn for every OpenRgn.

Here's an example of how to create and open a region, define a barbell shape, close the region, and draw it:

```
Barbell := NewRgn;                          {make a new region}
PenSize(Ø,Ø);                               {don't draw it on the screen}
OpenRgn;                                    {begin collecting stuff}
    SetRect(tempRect,2Ø,2Ø,3Ø,5Ø);          {draw the left weight}
    FrameOval(tempRect);
    SetRect(tempRect,3Ø,3Ø,8Ø,4Ø);          {draw the bar}
    FrameRect(tempRect);
    SetRect(tempRect,8Ø,2Ø,9Ø,5Ø);          {draw the right weight}
    FrameOval(tempRect);
CloseRgn(Barbell);                          {we're done; save in barbell}
FillRgn(Barbell,Black);                     {draw it! }
DisposeRgn(Barbell)                         {we don't need you anymore...}
```

```
PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: INTEGER);
```

Mathematically moves the region a distance of dh horizontally and dv vertically; positive directions are right and down. The region retains its size and shape; it merely is moved on the coordinate plane.

```
PROCEDURE InsetRgn (rgn: RgnHandle; dh,dv: INTEGER);
```

Shrinks or grows the region. All points on the region boundary are moved inwards a distance of dv vertically and dh horizontally; if dh or dv is negative, the points are moved outwards in that direction. InsetRgn leaves the region "centered" at the same position, but moves the outline in (for positive values) or out (for negative values). InsetRgn of a rectangular region works just like InsetRect (see).

```
PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

Calculates the intersection of two regions and places the intersection in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn before you call SectRgn. The destRgn can be one of the source regions, if desired.

If the regions do not intersect, or one of the regions is the empty region, the destination is set to the empty region, which you can test with the EmptyRgn function.

If the regions were defined in different grafPorts, the mathematical intersection (from SectRgn) may not correspond to their physical intersection on the screen. To adjust a region into another port's coordinate system, see the LocalToGlobal and GlobalToLocal calls.

PROCEDURE UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

Calculates the union of two regions and places the union in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn begore you call UnionRgn. The destRgn can be one of the source regions, if desired.

If both regions are empty, the destination is set to the empty region, which you can test with the EmptyRgn function.

If the regions were defined in different grafPorts, the mathematical union (from UnionRgn) may not correspond to their physical union on the screen. To adjust a region into another port's coordinate system, see the LocalToGlobal and GlobalToLocal calls.

PROCEDURE DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

Subtracts srcRgnB from srcRgnA and places the difference in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn begore you call DiffRgn. The destRgn can be one of the source regions, if desired.

If the first source region is the empty region, the destination is set to the empty region, which you can test with the EmptyRgn function.

If the regions were defined in different grafPorts, the mathematical difference (from DiffRgn) may not correspond to their physical difference on the screen. To adjust a region into another port's coordinate system, see the LocalToGlobal and GlobalToLocal calls.

PROCEDURE XorRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

Calculates the difference between the union and the intersection of two regions and places the result in a third region. THIS DOES NOT CREATE THE DESTINATION REGION: you must use NewRgn to create the dstRgn begore you call XorRgn. The destRgn can be one of the source regions, if desired.

If the regions are coincident, the destination is set to the empty region, which you can test with the EmptyRgn function.

If the regions were defined in different grafPorts, the mathematical Xor (from XorRgn) may not correspond to their physical Xor on the screen. To adjust a region into another port's coordinate system, see the LocalToGlobal and GlobalToLocal calls.

MISSING PAGE

MISSING PAGE

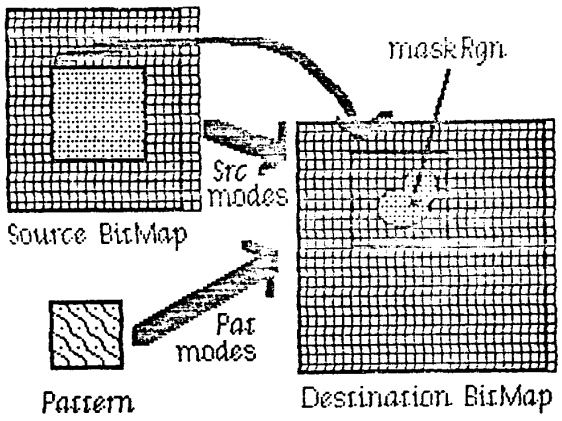


Figure 19. Operation of XferRgn

The DstRect and MaskRgn coordinates are in terms of the dstBits.bounds coordinate system; the SrcRect coordinates are in terms of the srcBits.bounds coordinates. When copying from the source to the destination, the top left corner of the source rectangle is aligned with the top left corner of the destination rectangle; the size of the source rectangle is ignored.

In the source ~~using~~ transfer modes, each bit of the source is transferred to the destination according to the rules of the mode. The pat parameter is ignored; we recommend you set it to white. The source ~~using~~ modes are as follows:

srcCopy	srcXor	notSrcCopy	notSrcXor
srcOr	srcBic	notSrcOr	notSrcBic

In the pattern ~~using~~ transfer modes, the specified pattern is transfered into the destination rectangle according to the chosen mode. The source bitMap and rectangle are not used; we suggest you set these to the destination bitMap and rectangle.

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

These pattern transfer modes are useful for filling a region that is in some other GrafPort.

Miscellaneous Utilities

FUNCTION Random: INTEGER;

Returns a 16-bit signed integer, uniformly distributed pseudo-random.

FUNCTION GetPixel (h,v: INTEGER): BOOLEAN;

Peeks at a single bit in memory (perhaps a pixel) and returns TRUE if the dot is black and FALSE if the dot is white. The selected pixel is the one immediately below and to the right of the point whose coordinates are given in h and v; these points are in the local coordinates of the current grafPort. There is no guarantee that the pixel actually belongs to the port, however; it may have been drawn by a port overlapping the current one. To see if the point indeed belongs to the current port, perform a PtInRgn(pt,thePort^.visRgn).

PROCEDURE AddPt (src: Point; VAR dst: Point);

Adds the first point to the second, returns the sum in the second point.

PROCEDURE SubPt (src: Point; VAR dst: Point);

Subtracts the first point from the second, returns the result in the second point.

PROCEDURE SetPt (VAR pt: Point; h,v: INTEGER);

Assigns two integer coordinates to a variable of type point.

PROCEDURE LocalToGlobal (VAR pt: Point);

Converts the given point from the current grafPort's local coordinate system into a global coordinate system with the origin (Ø,Ø) at the top left corner of the port's bitMap (such as the screen). This global point can then be compared to other global points, or be changed into the local coordinates of another grafPort. For example, if the rectangle ballRect is defined in the gamePort grafPort, and you want to use that rectangle in the selectPort grafPort, you would calculate the ball's coordinates like this:

```
SetPort(gamePort);                    { start in origin port}
SelectBall := ballRect;               {make a copy to be moved}
LocalToGlobal(SelectBall.topLeft);    { put both corners into }
LocalToGlobal(SelectBall.botRight);   {  global coordinates   }

SetPort(selectPort);                  { switch to destination port }
GlobalToLocal(SelectBall.topLeft);    { put both corners into     }
GlobalToLocal(Selectball.botRight);   {   these local coordinates }
FillOval(SelectBall,BallColor,8,8);   { Now you have the ball! }
```

You can also convert rectangles and regions into global coordinates without performing a LocalToGlobal call for each point. Simply perform LocalToGlobal for one point (for example, the top left corner of the region's bounding box), then use OffsetRect or OffsetRgn to move the rest of the object the distance of displacement:

```
CopyRgn(myRgn, tempRgn);
tempPoint := tempRgn^^.rgnBBox.topLeft;
LocalToGlobal(tempPoint);
OffsetRgn(tempRgn, tempPoint.h, tempPoint.v)
```

PROCEDURE GlobalToLocal

Takes a point expressed in global coordinates (with the top left corner of the bitMap as coordinate (Ø,Ø)) and converts it into the local coordinates of the current grafPort. The global point can be obtained with the LocalToGlobal call (see above).

You can convert global-coordinate rectangles and regions into local coordinates without performing a GlobalToLocal call for each point. Simply perform GlobalToLocal for one point (for example, the top left corner of the region's bounding box), then use OffsetRect or OffsetRgn to move the rest of the object the distance of displacement:

```
SetPort(MyNewPort);
tempPoint := globRgn^^.rgnBBox.topLeft;
GlobalToLocal(tempPoint);
OffsetRgn(globRgn, -tempPoint.h, -tempPoint.v)
```

PROCEDURE StuffHex (thingptr: WordPtr; s:Str255);

Pokes bits (expressed as a string of hexadecimal digits) into any data structure. This is a good way to create cursors, patterns, or bit images to be "stamped" onto the screen with XferRect. For example,

```
StuffHex(@stripes,'Ø1Ø2Ø4Ø81Ø2Ø4Ø8Ø')
```

places a striped pattern into the pattern variable stripes. Beware; there is no range checking on the size of the destination variable; it's easy to overrun the variable and trash something if you don't know what you're doing.

USING QUICKDRAW FROM ASSEMBLY LANGUAGE

All Macintosh User Interface ToolBox routines can be called from assembly-language programs as well as from Pascal. When you write an assembly-language program to use these routines, though, you must

emulate Pascal's parameter passing and variable transfer protocols.

The remainder of this section discusses how to use the QuickDraw constants, global variables, data types, procedures and functions from assembly language.

The primary aid to assembly-language programmers is a file named GRAFEQU.TEXT. If you .INCLUDE this file when you assemble your program, all the QuickDraw constants, offsets to locations of global variables, and offsets into the fields of structured types will be available in symbolic form to your programs.

Constants

QuickDraw constants are stored in the GRAFEQU.TEXT file, and you can use the constant values symbolically. For example, if you've loaded the effective address of the thePort^.chStyle.mode field into address register A2, you can set that field to the srcXor mode with this statement:

```
MOVE.W  #SRCXOR,(A2)
```

The only constants defined at this point are the sixteen transfer modes.

Data Types

Pascal's strong typing ability lets you write Pascal programs without really considering the size of a variable. But in assembly language, you must keep track of the size of every variable. The sizes of the standard Pascal data types are as follows:

Type	Size	
INTEGER	Word	(2 bytes)
LONGINT	Long	(4 bytes)
Pointer	Long	(4 bytes)
BOOLEAN	Word	(2 bytes)
REAL	Long	(4 bytes)
CHAR	Word	(2 bytes)

INTEGERs and LONGINTs are in two's complement form; BOOLEANS have their boolean value in bit 8 of the word (the low-order bit of the byte at the same location); CHARs are stored in the high-order byte of the word; and REALs are in the KCS standard format.

QuickDraw simple data types are constructed out of these fundamental types.

Type	Size	
WordPtr	Long	(4 bytes)
Str255	Page	(256 bytes)
Pattern		8 bytes
Bits16		32 bytes

Other data types are constructed as records of variables of the above types. The size of such a type is the sum of the sizes of all the fields in the record; the fields appear in the variable with the first field in the lowest address. For example, the data type BitMap, which is defined like this:

```
BitMap = RECORD
             baseAddr: WordPtr;
             rowBytes: INTEGER;
             bounds:   Rect
         END;
```

would be arranged in memory as seven words: a long for the baseAddr, a word for the rowBytes, and four words for the top, left, right, and bottom parts of the bounds rectangle. To assist you in referring to the fields inside a variable that has a structure like this, the GRAFEQU.TEXT file defines constants that you can use as offsets into the fields of a structured variable. For example, to move a bitMap's rowBytes value into D3, you would execute the following instruction:

```
MOVE.W  MYBITMAP+ROWBYTES,D3
```

Displacements are given in the GRAFEQU.TEXT file for all fields of all data types defined by QuickDraw.

To do double indirection (for example, to get at the top coordinate of a region's bounding box), you perform an LEA indirectly to obtain the effective address from the handle:

```
MOVE.L  MYHANDLE,A1             ; Load handle into A1
MOVE.L  (A1),A1                 ; Use handle to get pointer
MOVE.W  RGNSIZE+TOP(A1),D3      ; Load value using pointer
```

(eye)

For regions (and all other variable-length structures with handles), you must not move the pointer into a register once and just continue to use that pointer; you must do the double indirection each time. Every QuickDraw, ToolBox, or memory management call you make can possibly trigger a heap compaction that renders all pointers to movable heap items (like regions) invalid. The handles will remain valid, but pointers you've obtained through handles can be rendered invalid at any subroutine call or trap in your program.

Global Variables

Global variables are stored in a special section of Macintosh low memory; register A5 always points to this section of memory. The GRAFEQU.TEXT file defines a constant GRAFGLOB that points to the beginning of the QuickDraw variables in this space, and other constants to point at the individual variables. To access one of the variables, just sum the constants and index off of A5. For example, if you've are

preparing a bitMap record and you want that bitMap to share the same bit image as the screenBits global variable, you'd perform the following instruction:

```
MOVE.L  GRAFGLOB+SCREENBITS+BASEADDR(A5),MYBITMAP+BASEADDR
```

Procedures and Functions

To call a QuickDraw procedure or function, you must push all parameters to that function on the stack, then JSR to the function or procedure. When you link your program with QuickDraw, these JSRs are adjusted to refer to the jump table in low RAM, so that a JSR into the table redirects you to the actual location of the procedure or function. After the procedure or function has unstacked all its parameters and performed its operation, it returns to you with the stack cleared.

The only difficult part about calling QuickDraw procedures and functions is stacking the parameters. You must follow some strict rules:

- Save all registers you wish to preserve before you begin pushing parameters. Any QuickDraw procedure can destroy the contents of the registers AØ, A1, DØ, D1, and D2, but the others are never altered.
- Push the parameters in the order that they appear in the Pascal procedural interface.
- For integers, booleans, and characters, push a word; for pointers, handles, long integers, and reals, push a long.
- For any structured variable longer than four (4) bytes, push a pointer to the variable.
- For all VAR parameters, regardless of size, push a pointer to the variable.
- When calling a function, FIRST push a null entry equal to the size of the function result, THEN push all other parameters. The result will be left on the stack after the function returns to you.

This makes for a lengthy interface, but it also guarantees that you can mock up a Pascal version of your program, and when you later translate it into assembly code, it works the same. For example, the Pascal statement

```
Blackness := GetPixel(5Ø,MousePos.v);
```

would be written in assembly language like this:

```
CLR.W   -(SP)                   ; Save space for boolean result
MOVE.W  #32,-(SP)               ; Push constant 5Ø (decimal)
```

```
        MOVE.W  MOUSEPOS+V,-(SP)        ; Push the value of MOUSEPOS.V
        JSR     GETPIXEL                ; Call routine
        MOVE.W  (SP)+,BLACKNESS         ; Fetch result from stack.
```

This is a simple example, pushing and pulling word-long constants. Normally, you'll be pushing more pointers, using the PEA (Push Effective Address) instruction:

```
        FillRoundRect(MyRect,1,thePort^.penSize.v,White);

        PEA     MYRECT                  ; Push pointer to MYRECT
        MOVE.W  #1,-(SP)                ; Push constant 1
        MOVE.W  GRAFGLOB+THEPORT+PENSIZE+V(A5),-(SP)
                                        ; Push value of ThePort^.penSize.v
        PEA     GRAFGLOB+WHITE(A5)      ; Push pointer to global var White
        JSR     FILLROUNDRECT           ; Call the subroutine
```

Beware that the horizontal and vertical parts of a point do not appear in the same order in the data structure as they do in a procedure call! Even though you can move a point around with a MOVE.L instruction, you can't push that point as a parameter to a procedure which requires two integer coordinates. For example, to call MoveTo with a point argument, you must do two separate pushes:

```
        MOVE.W  MOUSEPOS+H,-(SP)        ; Push horizontal position
        MOVE.W  MOUSEPOS+V,-(SP)        ; Push vertical position
        JSR     MOVETO
```

Just doing a MOVE.L MOUSEPOS,-(SP) would transpose the vertical and horizontal results, and make MoveTo work funny.

SUMMARY OF QUICKDRAW

```
CONST srcCopy    = Ø;       { Destination := Source }
      srcOr      = 1;       { Destination := Source OR Destination }
      srcXor     = 2;       { Destination := Source XOR Destination }
      srcBic     = 3;       { Destination := Source BIC Destination }
      notSrcCopy = 4;       { Destination := NOT(Source)  }
      notSrcOr   = 5;       { Destination := NOT(Source) OR Destination }
      notSrcXor  = 6;       { Destination := NOT(Source) XOR Destination }
      notSrcBic  = 7;       { Destination := NOT(Source) BIC Destination }
      patCopy    = 8;       { Destination := Pattern  }
      patOr      = 9;       { Destination := Pattern OR Destination }
      patXor     = 1Ø;      { Destination := Pattern XOR Destination }
      patBic     = 11;      { Destination := Pattern BIC Destination }
      notPatCopy = 12;      { Destination := NOT(Pattern)  }
      notPatOr   = 13;      { Destination := NOT(Pattern) OR Destination }
      notPatXor  = 14;      { Destination := NOT(Pattern) XOR Destination }
      notPatBic  = 15;      { Destination := NOT(Pattern) BIC Destination }
```

BIC?

uppercase

```
TYPE WordPtr = ^INTEGER;
     Str255  = String[255];
     Pattern =  PACKED ARRAY[Ø..7] OF Ø..255;
     Bits16  =  ARRAY[Ø..15] OF INTEGER;
     VHSelect = (v,h);

     Point = RECORD CASE INTEGER OF

               Ø: (v: INTEGER;
                   h: INTEGER);

               1: (vh: ARRAY[VHSelect] OF INTEGER);

             END;

     Rect = RECORD CASE INTEGER OF

             Ø: (top:      INTEGER;
                 left:     INTEGER;
                 bottom:   INTEGER;
                 right:    INTEGER);

             1: (topLeft:  Point;
                 botRight: Point);
           END;

     BitMap = RECORD
                baseAddr: WordPtr;
                rowBytes: INTEGER;
                bounds:   Rect;
              END;

Cursor = RECORD
                data:    Bits16;
                mask:    Bits16;
                hotSpot: Point;
              END;

     CharStyle = RECORD
                   fontPtr:  Handle;
                   mode:     INTEGER;
                   bold:     INTEGER;
                   slant:    INTEGER;
                   extra:    INTEGER;
                   shadow:   INTEGER;
                 END;

     PenState = RECORD
                   pnLoc :    Point;
                   pnSize :   Point;
                   pnMode :   INTEGER;
                   pnPat :    Pattern
                 END;
```

```
RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;
Region    =  RECORD
               rgnSize:   INTEGER;  { rgnSize = 1Ø for rectangular }
               rgnBBox:   Rect;
               { more data if not rectangular }
             END;

GrafPtr  = ^GrafPort;
GrafPort = RECORD
             portBits:  BitMap;
             portRect:  Rect;
             visRgn:    RgnHandle;
             clipRgn:   RgnHandle;
             bkPat:     Pattern;
             chStyle:   Charstyle;
             pnLoc:     Point;
             pnSize:    Point;
             pnMode:    INTEGER;
             pnPat:     Pattern;
           END;


VAR thePort:    GrafPtr;
    white,black,gray,ltGray,dkGray: Pattern;
    arrow:      Cursor;
    grafError:  INTEGER;
    grafDebug:  BOOLEAN;
    screenBits: BitMap;
    charNormal: CharStyle;
    swapFont:   ProcPtr;
    rgnSave:    BOOLEAN;
```

Housekeeping and GrafPort Procedures

```
PROCEDURE InitGraf;
PROCEDURE OpenPort   (gp: GrafPtr);
PROCEDURE SetPort    (gp: GrafPtr);
PROCEDURE GetPort    (VAR gp: GrafPtr);
PROCEDURE SetPortBits(bm: BitMap);
PROCEDURE PortSize   (width,height: INTEGER);
PROCEDURE MovePortTo (leftGlobal,topGlobal: INTEGER);
PROCEDURE SetOrigin  (h,v: INTEGER);
PROCEDURE SetVisRgn  (rgn: RgnHandle);
PROCEDURE GetVisRgn  (VAR rgn: RgnHandle);
PROCEDURE SetClipRgn (rgn: RgnHandle);
PROCEDURE GetClipRgn (VAR rgn: RgnHandle);
PROCEDURE ClipRect   (r: Rect);
PROCEDURE BackPat    (pat: Pattern);
```

Cursor Handling

```
PROCEDURE SetCursor  (crsr: Cursor);
PROCEDURE HideCursor;
PROCEDURE ShowCursor;
PROCEDURE CursorVis  (visible: BOOLEAN);
```

Pen and Line Drawing

```
PROCEDURE GetPen      (VAR pt: Point);
PROCEDURE GetPenState (VAR pnState: PenState);
PROCEDURE SetPenState (pnState: PenState);
PROCEDURE PenSize     (width,height: INTEGER);
PROCEDURE PenMode     (mode: INTEGER);
PROCEDURE PenPat      (pat: Pattern);
PROCEDURE PenNormal;
PROCEDURE MoveTo      (h,v: INTEGER);
PROCEDURE Move        (dh,dv: INTEGER);
PROCEDURE LineTo      (h,v: INTEGER);
PROCEDURE Line        (dh,dv: INTEGER);
```

Character Drawing

```
PROCEDURE DrawChar    (ch: char);
PROCEDURE DrawString  (s: Str255);
PROCEDURE DrawText    (textBuf: WordPtr; firstByte,byteCount: INTEGER);
FUNCTION  CharWidth   (ch: CHAR): INTEGER;
FUNCTION  StringWidth (s: Str255): INTEGER;
FUNCTION  TextWidth   (textBuf: WordPtr; firstByte,byteCount: INTEGER): INTEGER;
PROCEDURE SetChar     (chS: CharStyle);
```

Calculations with Rectangles

```
PROCEDURE SetRect     (VAR r: Rect; left,top,right,bottom: INTEGER);
PROCEDURE OffsetRect  (VAR r: Rect; dh,dv: INTEGER);
PROCEDURE InsetRect   (VAR r: Rect; dh,dv: INTEGER);
FUNCTION  SectRect    (srcRectA,srcRectB: Rect; VAR dstRect: Rect): BOOLEAN;
FUNCTION  PtInRect    (pt: Point; r: Rect): BOOLEAN;
```

Graphical Operations on Rectangles

```
PROCEDURE FrameRect   (r: Rect);
PROCEDURE PaintRect   (r: Rect);
PROCEDURE EraseRect   (r: Rect);
PROCEDURE InvertRect  (r: Rect);
PROCEDURE FillRect    (r: Rect; pat: Pattern);
PROCEDURE ScrollRect  (dstRect: Rect; dh,dv: INTEGER; updateRgn: rgnHandle);
PROCEDURE XferRect    (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                       mode:INTEGER; pat:Pattern);
```

Graphical Operations on Ovals

```
PROCEDURE FrameOval  (r: Rect);
PROCEDURE PaintOval  (r: Rect);
PROCEDURE EraseOval  (r: Rect);
PROCEDURE InvertOval (r: Rect);
PROCEDURE FillOval   (r: Rect; pat: Pattern);
```

Graphical Operations on Rounded-Corner Rectangles

```
PROCEDURE FrameRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE PaintRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE EraseRoundRect  (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE InvertRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
PROCEDURE FillRoundRect   (r: Rect; ovalWidth,ovalHeight: INTEGER; pat: Pattern);
```

Calculations with Regions

```
FUNCTION  NewRgn:   RgnHandle;
PROCEDURE DisposeRgn(rgn: RgnHandle);
PROCEDURE CopyRgn   (srcRgn,dstRgn: RgnHandle);
PROCEDURE SetRectRgn(rgn: RgnHandle; left,top,right,bottom: INTEGER);
PROCEDURE RectRgn   (rgn: RgnHandle; r: Rect);
PROCEDURE OpenRgn;
PROCEDURE CloseRgn  (dstRgn: RgnHandle);
PROCEDURE OffsetRgn (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE InsetRgn  (rgn: RgnHandle; dh,dv: INTEGER);
PROCEDURE SectRgn   (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE UnionRgn  (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE DiffRgn   (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE XorRgn    (srcRgnA,srcRgnB,dstRgn: RgnHandle);
FUNCTION  PtInRgn   (pt: Point; rgn: RgnHandle): BOOLEAN;
FUNCTION  RectInRgn (r: Rect; rgn: RgnHandle): BOOLEAN;
FUNCTION  EqualRgn  (rgnA, rgnB: RgnHandle): BOOLEAN;
FUNCTION  EmptyRgn  (rgn: RgnHandle): BOOLEAN;
```

Graphical Operations on Regions

```
PROCEDURE FrameRgn  (rgn: RgnHandle);
PROCEDURE PaintRgn  (rgn: RgnHandle);
PROCEDURE EraseRgn  (rgn: RgnHandle);
PROCEDURE InvertRgn (rgn: RgnHandle);
PROCEDURE FillRgn   (rgn: RgnHandle; pat: Pattern);
PROCEDURE XferRgn   (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                     mode:INTEGER; pat:Pattern; maskRgn: RgnHandle);
```

Miscellaneous Utility Routines

```
FUNCTION  Random:        INTEGER;
FUNCTION  GetPixel       (h,v: INTEGER): BOOLEAN;
PROCEDURE AddPt          (src: Point; VAR dst: Point);
PROCEDURE SubPt          (src: Point; VAR dst: Point);
PROCEDURE SetPt          (VAR pt: Point; h,v: INTEGER);
PROCEDURE LocalToGlobal  (VAR pt: Point);
```

```
PROCEDURE GlobalToLocal  (VAR pt: Point);
PROCEDURE StuffHex       (thingptr: WordPtr; s:Str255);
```

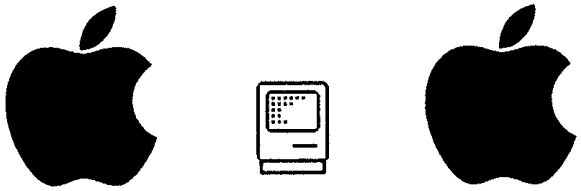
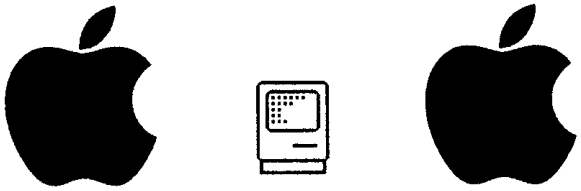
DEFAULT VALUES

These default values are set when you initialize MacGraf with the InitGraf procedure:

Global	Type	Setting
white	Pattern	All-white pattern
black	Pattern	All-black pattern
gray	Pattern	50% grey pattern
ltGray	Pattern	33% grey pattern
dkGray	Pattern	66% grey pattern
arrow	Cursor	Pointing arrow cursor
charNormal	charStyle	NO FONT, OR mode, no extras
screenBits	BitMap	Macintosh screen, (0,0,384,256)
thePort	GrafPtr	NIL
rgnSave	BOOLEAN	FALSE

These default values for a single GrafPort are set by OpenPort:

Variable	Type	Setting
portBits	BitMap	ScreenBits (see InitGraf)
portRect	Rect	ScreenBits.bounds (0,0,384,256)
visRgn^^	Region	Rectangular, (0,0,384,256)
clipRgn^^	Region	Rectangular, (-30000,-30000,30000,30000)
bkPat	Pattern	White
chStyle	CharStyle	CharNormal
pnLoc	Point	(0,0)
pnSize	Point	(1,1)
pnMode	INTEGER	patCopy (8)
pnPat	Pattern	Black



END OF DOCUMENT